Informatique tronc commun

Révisions sur les graphes

Poly de cours

1 Introduction 2
1.1 Le programme 2
1.2 Aux concours 2
1.3 L'idée du chapitre 3
2 Vocabulaire et représentations 4
2.1 Les graphes 4
2.2 Quelques graphes particuliers 5
2.3 Quatre représentations 6
2.4 Graphes pondérés 8
2.5 Passer d'une représentation à une autre 9
2.6 Les graphes qu'on ne construit pas 11
3 Parcourir un graphe 14
3.1 Un seul schéma 14
3.2 Le parcours en profondeur 16
3.3 Le parcours en largeur 18
3.4 L'arbre de parcours 20
4 Ce qu'on fait avec un parcours 20
4.1 Connexité et composantes connexes 20
4.2 Distances et plus courts chemins en nombre d'arêtes 22
4.3 Détecter un cycle dans un graphe non orienté 24
5 Plus courts chemins pondérés : l'algorithme de Dijkstra 25
5.1 Le problème 25
5.2 L'idée 26
5.3 Pourquoi le choix est sûr 26
5.4 Exécution à la main 27
5.5 Le code 28
5.6 Variantes 29
5.7 Les limites de l'algorithme 32
5.8 L'algorithme A* (HP) 33
5.9 Quel algorithme pour quel problème ? 36
6 Graphes orientés sans cycle 36
6.1 Détecter un cycle (HP) 36
6.2 Ordre topologique (HP) 37
6.3 Le lien avec la programmation dynamique 38

1 Introduction

1.1 Le programme

Les graphes sont au programme de première année. Ce chapitre effectue des **révisions** sur ce sujet, car les graphes apparaissent en deuxième année dans la programmation dynamique et dans l'étude des jeux.

2.3 Bases des graphes, plus courts chemins

Il s'agit de définir le modèle des graphes, leurs représentations et leurs manipulations.
On s'efforce de mettre en avant des applications importantes et si possible modernes : réseau de transport, graphe du web, réseaux sociaux, bio-informatique. On précise autant que possible la taille typique de tels graphes.

Notions	Commentaires
Vocabulaire des graphes.	Graphe orienté, graphe non orienté. Sommet (ou nœud) ; arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité dans les graphes non orientés. On présente l'implémentation des graphes à l'aide de listes d'adjacence (rassemblées par exemple dans une liste ou dans un dictionnaire) et de matrice d'adjacence. On n'évoque ni multi-arcs ni multi-arêtes.
Notations.	Graphe $G = (S, A)$, degrés $d(s)$ (pour un graphe non orienté), $d_+(s)$ et $d_-(s)$ (pour un graphe orienté).
Pondération d'un graphe. Étiquettes des arcs ou des arêtes d'un graphe.	On motive l'ajout d'information à un graphe par des exemples concrets.
Parcours d'un graphe.	On introduit à cette occasion les piles et les files ; on souligne les problèmes d'efficacité posés par l'implémentation des files par les listes de Python et l'avantage d'utiliser un module dédié tel que `collections.deque`. Détection de la présence de cycles ou de la connexité d'un graphe non orienté.
Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs.	Algorithme de Dijkstra. On peut se contenter d'un modèle de file de priorité naïf pour extraire l'élément minimum d'une collection. Sur des exemples, on s'appuie sur l'algorithme A* vu comme variante de celui de Dijkstra pour une première sensibilisation à la notion d'heuristique.

extrait programme ITC, 1re année

1.2 Aux concours

Dans les sujets d'ITC des Mines et de X-ENS, c'est souvent un « graphe déguisé » qui est présenté, la représentation étant fournie. Remarquer que les graphes peuvent aussi tomber dans un sujet de maths à l'oral … ou même à l'écrit !

Sujet	Le graphe	Ce qui est demandé
X-ENS 2016	réseau social, donné par la liste de ses arêtes	écrire les fonctions de base (arête ? voisins ?) avec leur complexité, structure de partition (représentant, fusion), coupe minimale par un algorithme aléatoire
X-ENS 2019	grille de cases colorées	taille d'une région unicolore par une fonction récursive, terminaison, critique d'un code fourni
Mines 2022	grille de spins	voisins d'une case, exploration récursive d'un domaine, puis avec une pile, numérotation des domaines
X-ENS 2023	grille implicite, décrite par une fonction `successeurs` à écrire	Dijkstra fourni : dire ce qu'il calcule, donner sa complexité, heuristique pour A*
Mines 2024	graphe en couches pondéré par des probabilités	compter sommets et arêtes, dessiner, se ramener à un plus court chemin, puis programmation dynamique
X-ENS 2025	grille avec sauts	chemins, absence de cycle, mémoïsation, programmation dynamique
X-ENS 2026	dictionnaire de listes d'adjacence, graphe orienté	parcours en profondeur fourni : le dérouler à la main, prouver ses propriétés, compter des chemins

Et dans des épreuves écrites de **mathématiques** :

Sujet	Le graphe	Ce qui est demandé
Centrale 2016, maths 1 (MP)	graphe orienté associé à une matrice positive (le mot « graphe » n'est pas écrit)	chemins et coefficients de A^m, matrices irréductibles et primitives
ENS 2016, maths D	graphe non orienté et sa matrice d'adjacence	constante de Cheeger : isoler une partie du graphe en coupant peu d'arêtes
Centrale 2021, maths 1 (PSI)	marche aléatoire sur un graphe (tétraèdre, cube)	matrices stochastiques, convergence des puissances, PageRank
Mines 2024, maths 2 (MP-MPI)	graphes aléatoires, matrice d'adjacence	rang et polynôme caractéristique d'une matrice d'adjacence, seuil d'apparition d'un sous-graphe
Centrale 2025, maths 2 (MP-MPI)	modèle d'Ising : particules en interaction avec leurs voisines, sur un cycle puis sur une grille	écrire en Python `mat_adj(graphe)` à partir d'un dictionnaire de listes d'adjacence, valeurs propres de la matrice d'adjacence

Il faut donc savoir:

- **Reconnaître un problème de graphe** (accessibilité, plus court chemin) dans une situation qui n'en parle pas.
- **Lire une représentation** décrite par l'énoncé et écrire deux ou trois fonctions élémentaires dessus.
- **Comprendre un code fourni** : le dérouler à la main, prouver une propriété, le compléter, le critiquer.
- **Donner une complexité**, en fonction du nombre de sommets et du nombre d'arêtes.

1.3 L'idée du chapitre

Il n'y a dans le fond qu'un seul algorithme de parcours. On range les sommets à visiter dans un « sac » :

- **si le « sac » est une pile, c'est le parcours en profondeur,**
- **si le « sac » est une file, c'est le parcours en largeur,**
- **si le « sac » est une file de priorité, c'est l'algorithme de Dijkstra.**

Voici ce parcours, écrit avec trois fonctions qui cachent la nature du sac : `sac_vide()` le crée, `ajouter(sac, s)` y range un sommet, `retirer(sac)` en sort un sommet et le renvoie.

```
def parcours_sac(G, depart):
    """Sommets accessibles depuis depart, dans l'ordre où ils sortent du sac."""
    vus = [False for s in range(len(G))]
    ordre = []
    sac = sac_vide()
    ajouter(sac, depart)
    while len(sac) > 0:
        s = retirer(sac)
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                ajouter(sac, t)
    return ordre
```

C'est un vrai code Python : il suffit de changer les fonctions du sac pour changer de parcours (partie 3). Les sommets déjà visités sont gérés par la liste `vus` : un sommet peut entrer plusieurs fois dans le sac, mais il n'est traité qu'à sa première sortie.

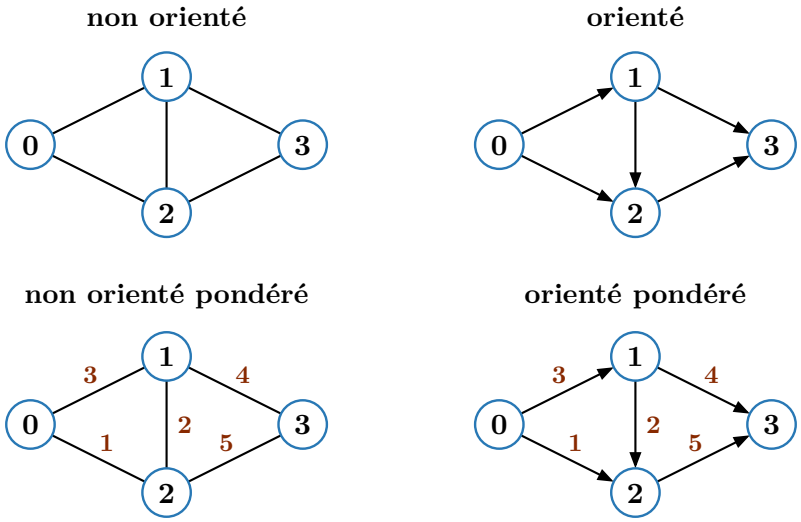
Presque tous les problèmes classiques se résolvent alors en ajoutant **une information** à ce parcours : un prédécesseur, une distance, un numéro de composante, etc.

2 Vocabulaire et représentations

2.1 Les graphes

Définition. Un **graphe** est un couple $G = (S, A)$ où S est un ensemble fini de **sommets** et A un ensemble d'**arêtes**.

- Dans un graphe **non orienté**, une arête est une paire $\{s, t\}$ de sommets distincts.
- Dans un graphe **orienté**, une arête est un couple (s, t) de sommets distincts : elle va de s vers t.
- Un graphe **pondéré**, orienté ou non, est un graphe muni d'une fonction $p : A \to \mathbb{R}$: le nombre $p(a)$ est le **poids** de l'arête a (une distance, une durée, un coût).



Le même graphe à quatre sommets, sous ses quatre formes.

Dans tout le chapitre, $n = |S|$ est le nombre de sommets et $m = |A|$ le nombre d'arêtes. On dit « arête » dans les deux cas (le programme dit aussi « arc » pour un graphe orienté). On exclut ici les **boucles**, arêtes d'un sommet vers lui-même.

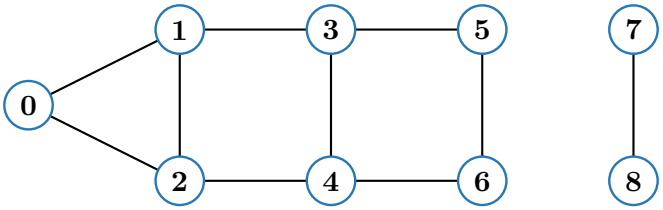
- Deux sommets reliés par une arête sont **voisins** (ou adjacents).
- Le **degré** $d(s)$ d'un sommet est le nombre de ses voisins. Dans un graphe orienté, on distingue le degré sortant $d^+(s)$ (arêtes qui partent de s) et le degré entrant $d^-(s)$ (arêtes qui arrivent en s).
- Un **chemin** de s à t est une suite de sommets $s = s_0, s_1, ..., s_k = t$ où chaque s_i est relié à s_{i+1} par une arête (orientée de s_i vers s_{i+1} dans un graphe orienté). Sa **longueur** est son nombre d'arêtes, k.
- Un **cycle** est un chemin de longueur au moins 1 d'un sommet à lui-même, sans arête répétée (dans un graphe non orienté, l'aller-retour s, t, s n'est donc pas un cycle).
- Un graphe non orienté est **connexe** si deux sommets quelconques sont reliés par un chemin. En général, « être relié par un chemin » est une relation d'équivalence dont les classes sont les **composantes connexes**.

Le programme ne parle de connexité que pour les graphes non orientés. Dans un graphe orienté, « il existe un chemin de s à t » n'est pas une relation symétrique, donc pas une relation d'équivalence : il faut demander un chemin dans chaque sens, ce qui s'appelle la forte connexité et n'est pas au programme.

Dans un graphe pondéré :

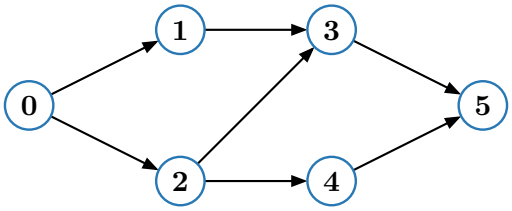
- le **poids** d'un chemin est la somme des poids de ses arêtes (supposés ici positifs ou nuls),
- la **distance** de s à t est le plus petit poids d'un chemin de s à t, et $+\infty$ s'il n'y en a pas, un chemin qui réalise ce minimum est un **plus court chemin**.

Dans un graphe non pondéré, la distance est le plus petit nombre d'arêtes d'un chemin : c'est le cas où tous les poids valent 1.



Exemple $n = 9$, $m = 10$, deux composantes connexes.

Dans ce graphe, $d(3) = 3$, le chemin $0, 2, 4, 6$ est de longueur 3, et $0, 1, 2, 0$ est un cycle.



Un graphe orienté : $n = 6$, $m = 7$.

Dans ce graphe orienté, $d^+(2) = 2$ et $d^-(2) = 1$, $0, 2, 4, 5$ est un chemin, mais il n'y a aucun chemin de 5 à 0, et aucun cycle.

On remarque que si un graphe est connexe $m \geq n - 1$ et que dans un graphe non orienté $m \leq \binom{n}{2} = \frac{n(n-1)}{2}$. De manière générale on a $m = O(n^2)$ mais dans des graphes peu denses, on peut aussi avoir $m = O(n)$. (La densité est le rapport $\frac{m}{\binom{n}{2}}$.)

Pour calculer les complexités il faut les exprimer a priori en **fonction de n et m**.

Pour cela le lemme essentiel est le suivant:

Lemme des poignées de main. Dans un graphe non orienté, $\sum_{s \in S} d(s) = 2m$. Dans un graphe orienté, $\sum_{s \in S} d^+(s) = \sum_{s \in S} d^-(s) = m$.

En effet, chaque arête est comptée une fois par chacune de ses deux extrémités.

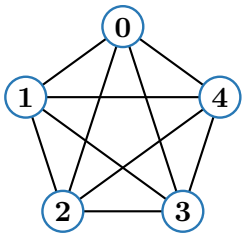
D'un point de vue mathématique, c'est un argument de parité. Par exemple, **dans un graphe non orienté, le nombre de sommets de degré impair est pair** : la somme de tous les degrés vaut $2m$, elle est paire, donc elle contient un nombre pair de termes impairs. Dans le graphe non orienté de l'exemple, les sommets de degré impair sont $1, 2, 3, 4, 7, 8$: il y en a six. Ainsi, dans une soirée, le nombre de personnes qui ont serré un nombre impair de mains est pair.

Mais ce lemme est crucial en termes de complexité : **parcourir les voisins de tous les sommets coûte $O(n + m)$, et non $O(n^2)$**.

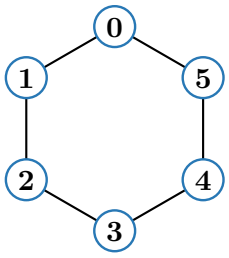
2.2 Quelques graphes particuliers

- Le **graphe complet** K_n a n sommets, tous reliés deux à deux : $m = \binom{n}{2}$, c'est le maximum.
- Le **cycle** C_n a n sommets disposés en cercle, chacun relié au suivant : $m = n$, et tous les degrés valent 2.
- Un **arbre** est un graphe connexe et sans cycle. Il a exactement $m = n - 1$ arêtes : c'est le minimum pour un graphe connexe (partie 4).

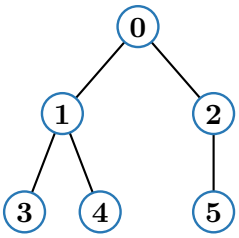
K_5 : graphe complet



C_6 : cycle



un arbre

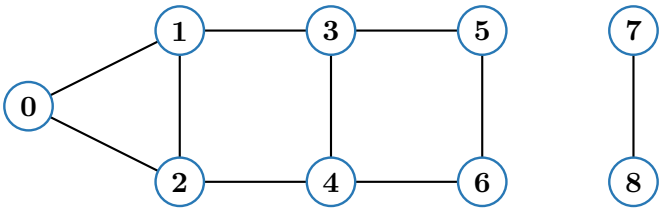


2.3 Quatre représentations

On numérote les sommets de 0 à $n-1$.

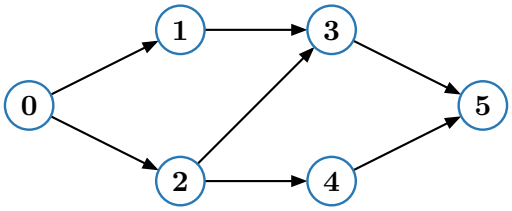
Listes d'adjacence. Le graphe est une liste `G` de longueur n : `G[s]` est la liste des voisins du sommet `s` (dans un graphe orienté, la liste des sommets `t` tels qu'il y a une arête de `s` vers `t`).

C'est **la représentation de référence de ce cours**.




```
G = [[1, 2],        # voisins de 0
     [0, 2, 3],     # voisins de 1
     [0, 1, 4],     # voisins de 2
     [1, 4, 5],     # voisins de 3
     [2, 3, 6],     # voisins de 4
     [3, 6],        # voisins de 5
     [4, 5],        # voisins de 6
     [8],           # voisins de 7
     [7]]           # voisins de 8
```

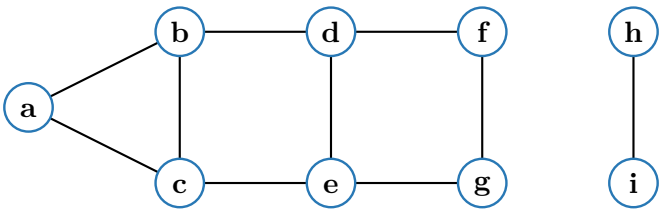
Dans un graphe non orienté, chaque arête apparaît **deux fois** : `2` est dans `G[4]` et `4` est dans `G[2]`. Pour un graphe orienté, `G[s]` ne contient que les sommets vers lesquels part une arête :




```
G_ORIENTE = [[1, 2], [3], [3, 4], [5], [5], []]
```

Le sommet 5, d'où ne part aucune arête, a une liste vide.

Dictionnaire d'adjacence. Quand les sommets ne sont pas des entiers (des noms de villes, des cases, des mots), on remplace la liste par un dictionnaire : `D[s]` est la liste des voisins de `s`.

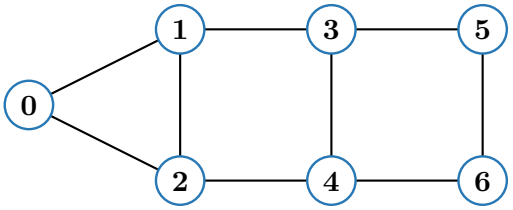


Le même graphe, avec des sommets nommés.

```
D = {"a": ["b", "c"],
     "b": ["a", "c", "d"],
     "c": ["a", "b", "e"],
     "d": ["b", "e", "f"],
     "e": ["c", "d", "g"],
     "f": ["d", "g"],
     "g": ["e", "f"],
     "h": ["i"],
     "i": ["h"]}
```

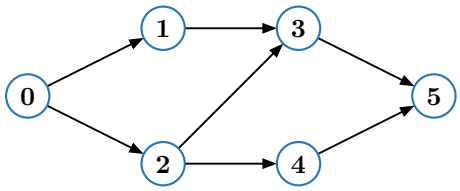
C'est la même idée : tous les codes du chapitre s'adaptent en remplaçant `range(len(G))` par un parcours des clés et les listes de marquage par des dictionnaires.

Matrice d'adjacence. Le graphe est une liste de listes `M` de taille $n \times n$: `M[s][t]` vaut `1` s'il y a une arête de `s` vers `t`, et `0` sinon. La matrice d'un graphe non orienté est symétrique.



	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0
2	1	1	0	0	1	0	0	0	0
3	0	1	0	0	1	1	0	0	0
4	0	0	1	1	0	0	1	0	0
5	0	0	0	1	0	0	1	0	0
6	0	0	0	0	1	1	0	0	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	1	0

Le graphe du cours : matrice symétrique.



	0	1	2	3	4	5
0	0	1	1	0	0	0
1	0	0	0	1	0	0
2	0	0	0	1	1	0
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	0

Le graphe orienté : la ligne `s` donne les arêtes qui partent de `s`.

```
M = [[0, 1, 1, 0, 0, 0, 0, 0, 0],
     [1, 0, 1, 1, 0, 0, 0, 0, 0],
     [1, 1, 0, 0, 1, 0, 0, 0, 0],
     [0, 1, 0, 0, 1, 1, 0, 0, 0],
     [0, 0, 1, 1, 0, 0, 1, 0, 0],
     [0, 0, 0, 1, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 1, 1, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 1],
     [0, 0, 0, 0, 0, 0, 0, 1, 0]]
```

Sur la ligne `s`, le nombre de `1` est le degré de `s` (le degré sortant pour un graphe orienté, le degré entrant se lit sur la colonne).

> La matrice d'adjacence est aussi un objet mathématique : le coefficient (s,t) de M^k est le nombre de chemins de longueur k de s à t. En particulier la trace de M^k compte les chemins fermés de longueur k : pour un graphe non orienté, la trace de M^2 vaut $2m$ et celle de M^3 vaut six fois le nombre de triangles (ici 6, pour le seul triangle $0, 1, 2$).

Liste d'arêtes. Le graphe est donné par son nombre de sommets et la liste de ses arêtes, chacune écrite une seule fois. C'est souvent sous cette forme qu'un graphe est **donné** (fichier, table SQL), rarement celle sous laquelle on le **traite**.

```
N = 9
ARETES = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 4),
          (3, 4), (3, 5), (4, 6), (5, 6), (7, 8)]
```

Il faut donner le nombre de sommets à part : un sommet sans aucune arête n'apparaît pas dans la liste.

Les quatre représentations contiennent la même information, mais les opérations de base n'y ont pas le même coût.

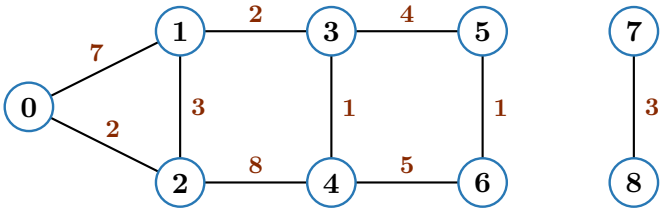
	Listes d'adjacence	**Dictionnaire**	**Matrice**	**Liste d'arêtes**
Mémoire	$O(n+m)$	$O(n+m)$	$O(n^2)$	$O(m)$
`s` et `t` voisins ?	$O(d(s))$	$O(d(s))$	$O(1)$	$O(m)$
Parcourir les voisins de `s`	$O(d(s))$	$O(d(s))$	$O(n)$	$O(m)$
Parcourir toutes les arêtes	$O(n+m)$	$O(n+m)$	$O(n^2)$	$O(m)$
Degré de `s`	$O(1)$	$O(1)$	$O(n)$	$O(m)$
Ajouter une arête	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Supprimer l'arête entre `s` et `t`	$O(d(s)+d(t))$	$O(d(s)+d(t))$	$O(1)$	$O(m)$
Ajouter un sommet	$O(1)$	$O(1)$	$O(n)$	$O(1)$

Le degré se lit par `len(G[s])`. Ajouter une arête coûte $O(1)$ si l'on ne vérifie pas qu'elle n'y est pas déjà. Ajouter un sommet à une matrice demande une case de plus sur chaque ligne et une ligne de plus.

Un parcours passe son temps à énumérer les voisins d'un sommet : les listes d'adjacence sont faites pour cela. La matrice est meilleure pour tester une arête, mais elle coûte $O(n^2)$ en mémoire et chaque énumération de voisins y coûte $O(n).$

2.4 Graphes pondérés

Le poids se range à côté du voisin : dans la liste d'adjacence, `G[s]` devient la liste des couples `(voisin, poids)`. Voici le graphe du cours avec des poids, qui servira pour l'algorithme de Dijkstra.




```
G_POIDS = [[(1, 7), (2, 2)], [(0, 7), (2, 3), (3, 2)], [(0, 2), (1, 3), (4, 8)],
           [(1, 2), (4, 1), (5, 4)], [(2, 8), (3, 1), (6, 5)], [(3, 4), (6, 1)],
           [(4, 5), (5, 1)], [(8, 3)], [(7, 3)]]
```

On parcourt les voisins d'un sommet par `for (t, poids) in G_POIDS[s]`. Dans une matrice, `M[s][t]` contient le poids de l'arête, et une valeur convenue en l'absence d'arête : ici `INFINI`, défini une fois pour toutes par

```
INFINI = float("inf")
```

	0	1	2	3	4	5	6	7	8
0	∞	7	2	∞	∞	∞	∞	∞	∞
1	7	∞	3	2	∞	∞	∞	∞	∞
2	2	3	∞	∞	8	∞	∞	∞	∞
3	∞	2	∞	∞	1	4	∞	∞	∞
4	∞	∞	8	1	∞	∞	5	∞	∞
5	∞	∞	∞	4	∞	∞	1	∞	∞
6	∞	∞	∞	∞	5	1	∞	∞	∞
7	∞	∞	∞	∞	∞	∞	∞	∞	3
8	∞	∞	∞	∞	∞	∞	∞	3	∞

La matrice des poids du graphe du cours.

La liste d'arêtes devient une liste de triplets `(s, t, poids)` :

```
ARETES_POIDS = [(0, 1, 7), (0, 2, 2), (1, 2, 3), (1, 3, 2), (2, 4, 8),
                (3, 4, 1), (3, 5, 4), (4, 6, 5), (5, 6, 1), (7, 8, 3)]
```

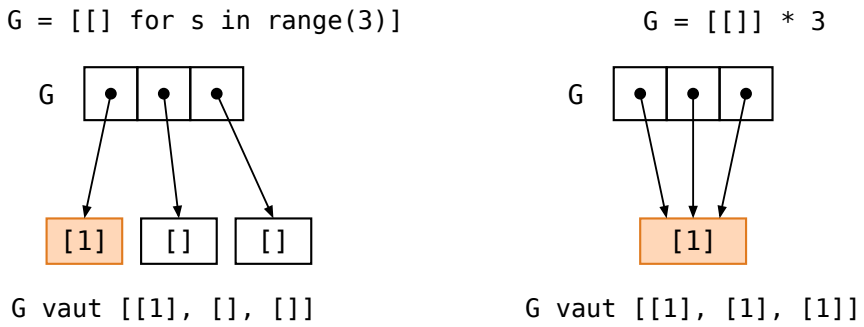
2.5 Passer d'une représentation à une autre

Ce sont les questions de début de sujet. Toutes suivent le même schéma : créer la structure vide, puis la remplir en parcourant l'autre.

```
def aretes_vers_liste(n, aretes):
    """Listes d'adjacence du graphe non orienté à n sommets d'arêtes données."""
    G = [[] for s in range(n)]
    for (s, t) in aretes:
        G[s].append(t)
        G[t].append(s)
    return G
```

Pour un graphe orienté, on ne garde que `G[s].append(t)`.

Attention. `G = [[]] * n` ne crée pas n listes vides mais n fois **la même** liste : après `G[0].append(1)`, toutes les cases contiennent `[1]`. Il faut une compréhension, `[[] for s in range(n)]`.



Après `G[0].append(1)` : à gauche trois listes distinctes, à droite trois fois la même.

```
def liste_vers_aretes(G):
    """Chaque arête {s, t} du graphe non orienté, une seule fois, avec s < t."""
    aretes = []
    for s in range(len(G)):
        for t in G[s]:
            if s < t:
                aretes.append((s, t))
    return aretes
```

```python
def liste_vers_matrice(G):
    n = len(G)
    M = [[0 for t in range(n)] for s in range(n)]
    for s in range(n):
        for t in G[s]:
            M[s][t] = 1
    return M

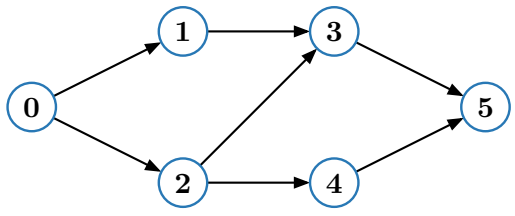
def matrice_vers_liste(M):
    n = len(M)
    G = [[] for s in range(n)]
    for s in range(n):
        for t in range(n):
            if M[s][t] == 1:
                G[s].append(t)
    return G
```

`liste_vers_matrice` coûte $O(n^2)$ (création de la matrice) et `matrice_vers_liste` aussi : avec une matrice, on ne descend jamais en dessous de n^2.

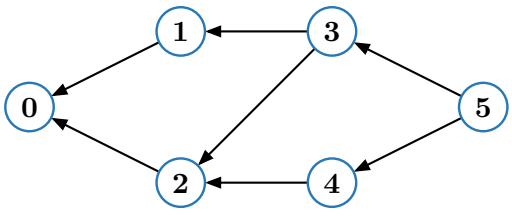
```python
def liste_vers_dico(G, noms):
    """Dictionnaire d'adjacence : le sommet s s'appelle noms[s]."""
    D = {}
    for s in range(len(G)):
        D[noms[s]] = [noms[t] for t in G[s]]
    return D

def dico_vers_liste(D):
    """Renvoie (G, noms) : on numérote les sommets dans l'ordre des clés."""
    noms = [nom for nom in D]
    numero = {}
    for s in range(len(noms)):
        numero[noms[s]] = s
    G = [[numero[nom] for nom in D[noms[s]]] for s in range(len(noms))]
    return G, noms
```

Deux constructions propres aux graphes orientés : le **graphe renversé**, où chaque arête est retournée, et le test de symétrie, qui dit si un graphe orienté est en fait un graphe non orienté.



`G_ORIENTE`



`renverse(G_ORIENTE)`

`renverse(G_ORIENTE)` vaut `[[], [0], [0], [1, 2], [2], [3, 4]]` : la liste de `s` contient maintenant les sommets d'où **arrive** une arête.

```python
def renverse(G):
    """Graphe orienté obtenu en retournant toutes les arêtes."""
    R = [[] for s in range(len(G))]
    for s in range(len(G)):
        for t in G[s]:
            R[t].append(s)
    return R

def est_voisin(G, s, t):
    for u in G[s]:
        if u == t:
            return True
    return False

def est_symetrique(G):
    """Un graphe orienté est-il la version orientée d'un graphe non orienté ?"""
    for s in range(len(G)):
        for t in G[s]:
            if not est_voisin(G, t, s):
                return False
    return True
```

Les versions pondérées s'écrivent de la même façon :

```python
def aretes_poids_vers_liste(n, aretes):
    """aretes : liste de triplets (s, t, poids), graphe non orienté."""
    G = [[] for s in range(n)]
    for (s, t, poids) in aretes:
        G[s].append((t, poids))
        G[t].append((s, poids))
    return G

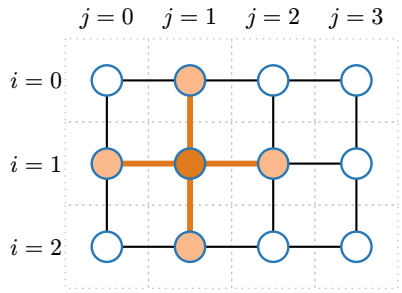
def liste_vers_matrice_poids(G):
    """M[s][t] est le poids de l'arête de s à t, INFINI s'il n'y en a pas."""
    n = len(G)
    M = [[INFINI for t in range(n)] for s in range(n)]
    for s in range(n):
        for (t, poids) in G[s]:
            M[s][t] = poids
    return M
```

2.6 Les graphes qu'on ne construit pas

Souvent le graphe n'est stocké nulle part : on sait seulement le découvrir, c'est-à-dire **calculer les voisins** d'un sommet.

Cela reste suffisant pour effectuer un **parcours**.

- **Une grille.** Les sommets sont les cases (i, j), deux cases sont voisines si elles se touchent par un côté.



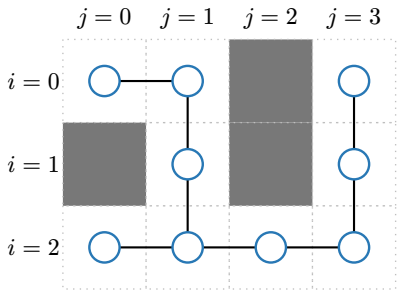
Une grille 3×4 : 12 sommets, 17 arêtes. En couleur, la case $(1,1)$ et ses quatre voisines.

```python
def voisins_grille(nb_lignes, nb_colonnes, i, j):
    """Cases voisines de la case (i, j) dans une grille."""
    voisins = []
    for (di, dj) in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        i_vois = i + di
        j_vois = j + dj
        if 0 <= i_vois and i_vois < nb_lignes:
            if 0 <= j_vois and j_vois < nb_colonnes:
                voisins.append((i_vois, j_vois))
    return voisins
```

`voisins_grille(3, 4, 1, 1)` renvoie `[(0, 1), (2, 1), (1, 0), (1, 2)]`, et `voisins_grille(3, 4, 0, 0)` seulement `[(1, 0), (0, 1)]`.

Deux variantes reviennent souvent.

- **Des cases interdites** (un labyrinthe, des obstacles) : `grille[i][j]` vaut `1` pour un mur et `0` pour une case libre. Les sommets sont les cases libres, et l'on ne garde que les voisines libres.
- **Des cases pondérées** : `grille[i][j]` est le coût à payer pour entrer dans la case (i,j). Le graphe est alors orienté et pondéré : l'arête de la case $(0,0)$ vers la case $(0,1)$ pèse `grille[0][1]`, celle du retour pèse `grille[0][0]`. On y revient avec l'algorithme de Dijkstra (partie 5).



`LABYRINTHE` : trois murs, 9 sommets, 8 arêtes.

1	9	1	1
1	9	1	9
1	1	1	1

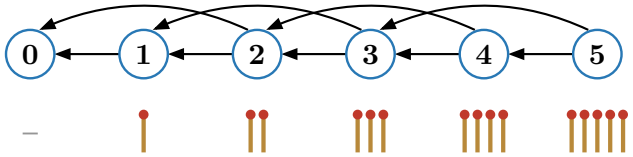
Une grille de coûts.

```python
LABYRINTHE = [[0, 0, 1, 0],
              [1, 0, 1, 0],
              [0, 0, 0, 0]]

def voisins_libres(grille, i, j):
    """Cases voisines de (i, j) qui ne sont pas des murs (un mur vaut 1)."""
    voisins = []
    for (i_vois, j_vois) in voisins_grille(len(grille), len(grille[0]), i, j):
        if grille[i_vois][j_vois] == 0:
            voisins.append((i_vois, j_vois))
    return voisins
```

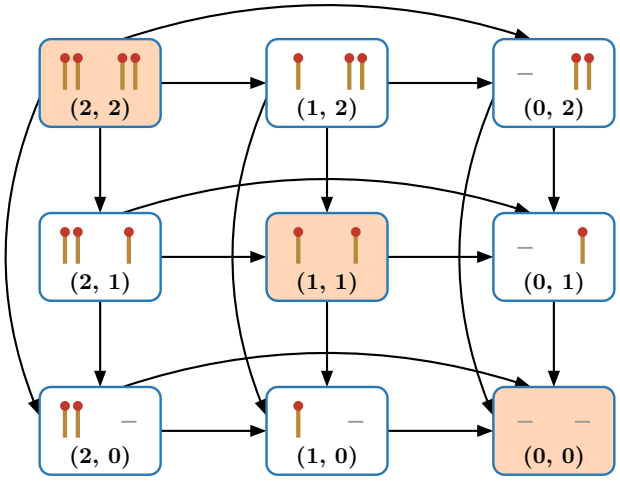
`voisins_libres(LABYRINTHE, 1, 1)` renvoie `[(0, 1), (2, 1)]`.

- **Les états d'un jeu ou d'un système.** Un sommet est une configuration (un plateau, une position et ce qu'on a ramassé), une arête est un coup. Le graphe est en général bien trop grand pour être construit, on l'explore à la demande. Un exemple minuscule : un tas de 5 allumettes, chaque joueur en retire une ou deux. Le sommet k est « il reste k allumettes ».



Le graphe du jeu des allumettes : sous chaque sommet, le tas correspondant.

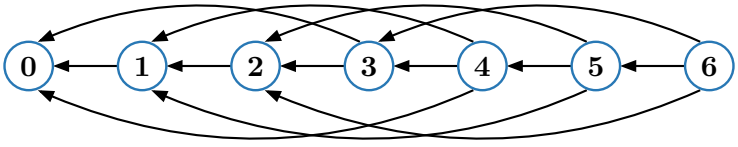
Avec plusieurs tas, c'est le **jeu de Nim** : à son tour, un joueur choisit un tas et y retire autant d'allumettes qu'il veut, au moins une. Un sommet est alors un couple (ou un k-uplet) de tailles de tas. Avec deux tas de deux allumettes, il y a 9 états et 18 arêtes : une flèche horizontale retire des allumettes du premier tas, une flèche verticale du second.



Le jeu de Nim à deux tas de deux allumettes.

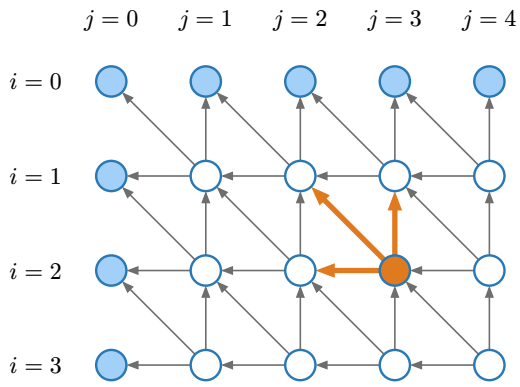
Le joueur qui prend la dernière allumette gagne. Les états colorés, où les deux tas sont égaux, sont ceux où le joueur qui doit jouer perd si son adversaire joue bien : quel que soit son coup, l'adversaire peut rendre les deux tas égaux. C'est le genre de question que l'on étudiera sur les graphes de jeux.

- **Les sous-problèmes d'une programmation dynamique.** Un sommet est un état, et il y a une arête de chaque état vers ceux dont son calcul dépend. Pour le rendu de monnaie avec les pièces 1, 3 et 4, le sommet k est « rendre la somme k » et dépend de $k-1$, $k-3$ et $k-4$.



Le graphe des sous-problèmes du rendu de monnaie.

Quand l'état est un couple, le graphe se dessine sur une grille. Pour la plus longue sous-suite commune de deux mots, l'état (i, j) désigne les préfixes de longueurs i et j, et $L(i, j)$ se calcule à partir de $L(i-1, j-1)$, $L(i-1, j)$ et $L(i, j-1)$: chaque sommet a au plus trois arêtes sortantes, vers le haut, vers la gauche et en diagonale.



Mots de longueurs 3 et 4 : l'état $(2,3)$ et ses trois arêtes, en bleu, les cas de base.

Remplir le tableau ligne par ligne, et chaque ligne de gauche à droite, c'est traiter les couples (i,j) dans l'**ordre lexicographique** : chaque sommet vient alors après ceux dont il dépend (partie 6).

On remarque que le graphe d'un jeu qui se termine toujours, comme celui des sous-problèmes d'une programmation dynamique, n'a pas de cycle.

Devant un énoncé, la première question est : quels sont les sommets, quelles sont les arêtes, et comment obtient-on les voisins d'un sommet ?

3 Parcourir un graphe

3.1 Un seul schéma

Parcourir un graphe depuis un sommet de départ, c'est visiter tous les sommets qu'on peut atteindre en suivant des arêtes, chacun une seule fois. Tous les parcours suivent le même schéma : on tient un **sac** de sommets à visiter, et on **marque** les sommets visités pour ne pas tourner en rond.

C'est le code de l'introduction :

```
def parcours_sac(G, depart):
    """Sommets accessibles depuis depart, dans l'ordre où ils sortent du sac."""
    vus = [False for s in range(len(G))]
    ordre = []
    sac = sac_vide()
    ajouter(sac, depart)
    while len(sac) > 0:
        s = retirer(sac)
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                ajouter(sac, t)
    return ordre
```

Il reste à écrire les trois fonctions : deux lignes changent entre la pile et la file.

Avec une pile

```
# le sac est une liste

def sac_vide():
    return []

def ajouter(sac, s):
    sac.append(s)

def retirer(sac):
    return sac.pop()
```

Avec une file

```
# on suppose deque importé

def sac_vide():
    return deque()

def ajouter(sac, s):
    sac.append(s)

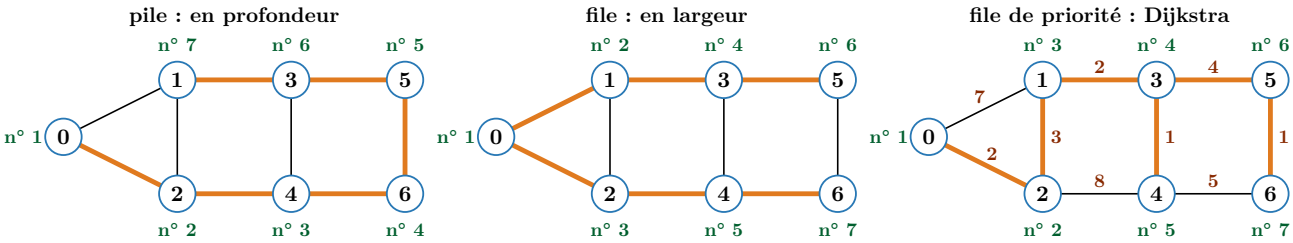
def retirer(sac):
    return sac.popleft()
```

Sur le graphe du cours, `parcours_sac(G, 0)` renvoie `[0, 2, 4, 6, 5, 3, 1]` avec la pile et `[0, 1, 2, 3, 4, 5, 6]` avec la file. Pour une file de priorité, `ajouter` et `retirer` changent toutes les deux : `ajouter` range le sommet avec une priorité, son estimation de distance au départ (il faut donc la lui transmettre), et `retirer` sort le sommet de plus petite priorité. C'est l'algorithme de Dijkstra, détaillé dans la partie 5.

Et les sommets déjà visités ? Rien n'empêche un sommet d'entrer plusieurs fois dans le sac, une fois par voisin qui le découvre. C'est le test `if not vus[s]`, fait **à la sortie**, qui garantit que chaque sommet n'est traité qu'une fois. Dans la suite, on allège le sac en n'y mettant pas les voisins déjà marqués, mais le test à la sortie reste nécessaire, sauf si l'on marque les sommets dès leur **entrée** dans le sac, comme on le fera pour la file (voir plus bas).

Le point important est la gestion du sac : quel élément en sort-on ?

Le sac est...	On sort...	On obtient...
une pile	le dernier entré	le parcours en profondeur
une file	le premier entré	le parcours en largeur
une file de priorité	le plus proche du départ	l'algorithme de Dijkstra



Les trois parcours depuis le sommet 0 : numéro de visite de chaque sommet, et en gras l'arbre du parcours.

Ce que calcule un parcours. Quel que soit le sac, à la fin du parcours les sommets marqués sont exactement les sommets **accessibles** depuis le départ, c'est-à-dire ceux vers lesquels il existe un chemin.

En effet, un sommet n'entre dans le sac que comme voisin d'un sommet marqué, donc tout sommet marqué est accessible. Réciproquement, s'il existait un sommet accessible non marqué, on prendrait sur un chemin qui y mène le premier sommet non marqué : son prédécesseur est marqué, donc il a été mis dans le sac, donc il en est sorti et a été marqué, ce qui est contradictoire.

L'**ensemble** des sommets visités ne dépend donc pas du sac. Ce qui en dépend, c'est l'**ordre** dans lequel ils sont visités, et c'est lui qui porte l'information utile : avec une file, les sommets sortent par nombre d'arêtes croissant depuis le départ, avec une file de priorité, par distance croissante, avec une pile, on suit un chemin jusqu'au bout avant d'en essayer un autre. L'ordre dépend aussi de l'ordre des voisins dans les listes d'adjacence : un énoncé qui demande « le » parcours en profondeur sous-entend qu'on suit l'ordre des listes.

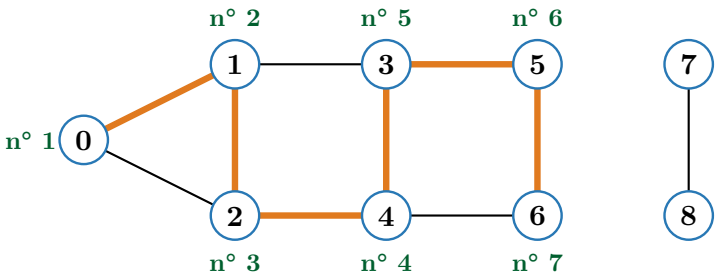
3.2 Le parcours en profondeur

On avance le plus loin possible, de voisin non vu en voisin non vu. Quand le sommet courant n'a plus de voisin non vu, on est bloqué : on **revient en arrière**, c'est-à-dire qu'on retourne au sommet depuis lequel on l'avait découvert, pour essayer son voisin suivant. La **pile** est alors celle des appels récursifs : c'est l'écriture la plus simple.

```python
def profondeur_aux(G, s, vus, ordre):
    vus[s] = True
    ordre.append(s)
    for t in G[s]:
        if not vus[t]:
            profondeur_aux(G, t, vus, ordre)

def parcours_profondeur(G, depart):
    """Sommets accessibles depuis depart, dans l'ordre du parcours."""
    vus = [False for s in range(len(G))]
    ordre = []
    profondeur_aux(G, depart, vus, ordre)
    return ordre
```

Les listes `vus` et `ordre` sont créées une fois par la fonction principale et **modifiées** par la fonction auxiliaire. Sur le graphe du cours, `parcours_profondeur(G, 0)` renvoie `[0, 1, 2, 4, 3, 5, 6]` : depuis 0 on va en 1, puis en 2 (premier voisin non vu de 1), puis en 4, 3, 5, 6, sans jamais avoir besoin de revenir en arrière.



`parcours_profondeur(G, 0)` : ordre de visite, et en gras les appels récursifs.

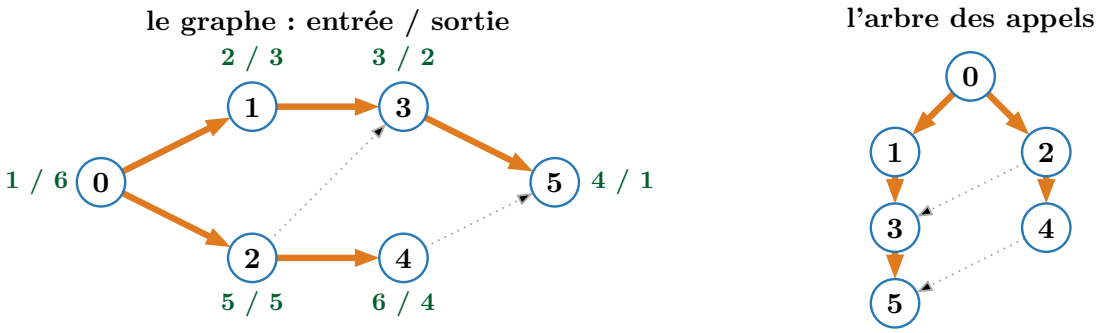
Terminaison et complexité. `profondeur_aux` est appelée au plus une fois par sommet, puisqu'elle commence par le marquer et qu'on ne l'appelle que sur un sommet non marqué. L'appel sur `s` coûte $O(1 + d(s))$ hors appels récursifs. Au total :

$$\sum_{s \in S} O(1 + d(s)) = O(n + m)$$

d'après le lemme des poignées de main. Avec une matrice d'adjacence, énumérer les voisins de `s` coûte $O(n)$ et le parcours coûte $O(n^2)$.

Revenir en arrière. Dans le code, revenir en arrière n'est écrit nulle part : c'est la fin d'un appel récursif. Quand la boucle `for t in G[s]` de l'appel sur `s` se termine, cet appel est fini et l'exécution reprend dans l'appel qui l'avait lancé, donc au sommet précédent, dont la boucle `for` continue avec le voisin suivant.

Sur le graphe du cours, le parcours depuis 0 visite tout en une seule descente, et les retours ne servent à rien. Sur le graphe orienté ci-dessous, il descend $0, 1, 3, 5$. Le sommet 5 n'a pas de voisin : son appel se termine et l'on revient en 3, qui n'a plus de voisin non vu, puis en 1, puis en 0. La boucle de 0 passe alors à son second voisin, 2, d'où l'on atteint 4. Les appels récursifs forment un arbre, les arêtes en pointillé mènent à un sommet déjà vu et ne provoquent pas d'appel.

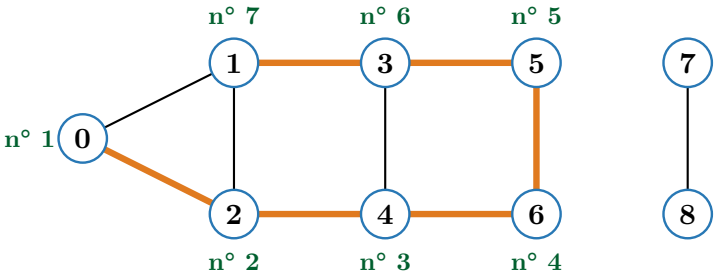


`parcours_profondeur(G_ORIENTE, 0)` renvoie `[0, 1, 3, 5, 2, 4]`. À côté de chaque sommet : son rang d'entrée dans le parcours, puis son rang de sortie (fin de son appel).

Sur un grand graphe, les appels récursifs imbriqués peuvent être trop nombreux pour Python : on remplace alors la récursivité par une pile explicite. Une pile est une liste Python dont on n'utilise que `append` (empiler) et `pop()` (dépiler le dernier), toutes deux en $O(1)$. C'est le schéma général, où l'on évite seulement d'empiler un voisin déjà marqué :

```
def parcours_pile(G, depart):
    """Parcours en profondeur sans récursivité : la pile est une liste."""
    vus = [False for s in range(len(G))]
    ordre = []
    pile = [depart]
    while len(pile) > 0:
        s = pile.pop()
        if not vus[s]:
            vus[s] = True
            ordre.append(s)
            for t in G[s]:
                if not vus[t]:
                    pile.append(t)
    return ordre
```

`parcours_pile(G, 0)` renvoie `[0, 2, 4, 6, 5, 3, 1]`. Ce n'est pas l'ordre de la version récursive, parce que le dernier voisin empilé est le premier visité, mais c'est bien un parcours en profondeur.



`parcours_pile(G, 0)`.

Voici le contenu de la pile au fil de cette exécution (le sommet de la pile est à droite).

On dépile	Pile ensuite	Commentaire
	`[0]`	au début
0	`[1, 2]`	on empile les voisins de 0
2	`[1, 1, 4]`	1 est empilé une seconde fois, 0 est déjà vu
4	`[1, 1, 3, 6]`	
6	`[1, 1, 3, 5]`	
5	`[1, 1, 3, 3]`	3 est empilé une seconde fois
3	`[1, 1, 3, 1]`	1 est empilé une troisième fois
1	`[1, 1, 3]`	tous ses voisins sont déjà vus
3, 1, 1	`[]`	déjà vus : on les dépile sans rien faire

Un même sommet peut donc se trouver plusieurs fois dans la pile, d'où le test `if not vus[s]` à la sortie. Chaque arête fait empiler au plus un sommet dans chaque sens, donc la pile reçoit au plus $1 + 2m$ éléments : la complexité reste $O(n + m)$.

3.3 Le parcours en largeur

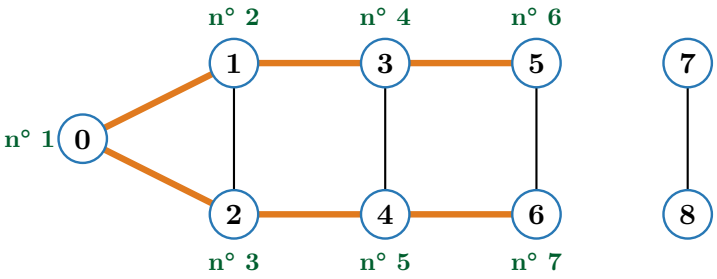
On visite d'abord les voisins du départ, puis les voisins des voisins, et ainsi de suite : le sac est une **file**, premier entré, premier sorti.

Attention. Une liste Python fait une mauvaise file : retirer son premier élément oblige à décaler tous les autres, ce qui coûte un temps proportionnel à la longueur de la liste. On utilise le type `deque` du module `collections`.

`from collections import deque`	à écrire une fois en tête de programme
`file = deque()`	crée une file vide
`file.append(x)`	ajoute `x` à la fin de la file, en $O(1)$
`file.popleft()`	retire et renvoie l'élément en tête de file, en $O(1)$
`len(file)`	nombre d'éléments de la file

```python
def parcours_largeur(G, depart):
    """Sommets accessibles depuis depart, dans l'ordre du parcours."""
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    vus[depart] = True
    while len(file) > 0:
        s = file.popleft()
        ordre.append(s)
        for t in G[s]:
            if not vus[t]:
                vus[t] = True
                file.append(t)
    return ordre
```

`parcours_largeur(G, 0)` renvoie `[0, 1, 2, 3, 4, 5, 6]`.



`parcours_largeur(G, 0)` : les sommets sont visités colonne par colonne.

On défile	File ensuite	Commentaire
	`[0]`	au début
0	`[1, 2]`	les voisins de 0 entrent, et sont marqués
1	`[2, 3]`	2 est déjà marqué : il n'entre pas une seconde fois
2	`[3, 4]`	
3	`[4, 5]`	
4	`[5, 6]`	
5	`[6]`	
6	`[]`	

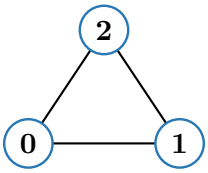
La tête de la file est à gauche. La complexité est encore $O(n+m)$: chaque sommet entre une seule fois dans la file, et à sa sortie on parcourt sa liste de voisins.

On marque un sommet au moment où il entre dans la file, pas au moment où il en sort.

Ainsi chaque sommet n'entre qu'une fois, la file contient au plus n éléments, et le test à la sortie devient inutile. Marquer à la sortie n'est pas faux si l'on garde le test du schéma général, mais oublier ce test donne un code qui visite plusieurs fois les mêmes sommets :

```
def parcours_largeur_maladroit(G, depart):
    """Parcours en largeur mal écrit (marquage à la sortie, sans test)."""
    vus = [False for s in range(len(G))]
    ordre = []
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        vus[s] = True               # marqué trop tard,
        ordre.append(s)             # et sans vérifier qu'il ne l'était pas déjà
        for t in G[s]:
            if not vus[t]:
                file.append(t)
    return ordre
```

Sur le graphe du cours, ce code renvoie `[0, 1, 2, 2, 3, 4, 4, 4, 5, 6, 6, 6, 6]` : le sommet 2 est mis dans la file par 0, puis de nouveau par 1 puisqu'il n'est pas encore marqué. Pour critiquer un code de parcours, on cherche un sommet atteint par deux chemins : un triangle suffit.



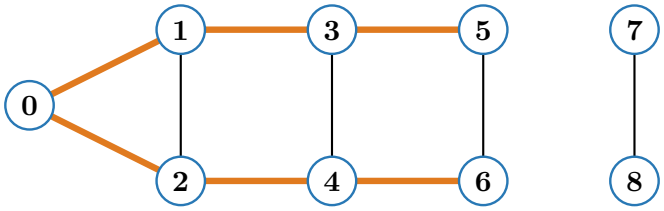
Sur le triangle `[[1, 2], [0, 2], [0, 1]]`, depuis 0, le code maladroit renvoie `[0, 1, 2, 2]`.

3.4 L'arbre de parcours

Pendant un parcours, chaque sommet autre que le départ est découvert **depuis** un autre sommet. Il suffit de retenir lequel :

```
def arbre_largeur(G, depart):
    """pred[t] : sommet depuis lequel t a été découvert (-1 si t non atteint)."""
    pred = [-1 for s in range(len(G))]
    pred[depart] = depart
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        for t in G[s]:
            if pred[t] == -1:
                pred[t] = s
                file.append(t)
    return pred
```

La liste `pred` joue aussi le rôle du marquage (`-1` signifie « pas encore vu »), et par convention le départ est son propre prédécesseur. Les arêtes `(pred[t], t)`, pour `t` atteint autre que le départ, forment un **arbre**, c'est-à-dire un graphe connexe et sans cycle : ce sont les arêtes en gras des figures. Il relie les sommets atteints avec le moins d'arêtes possible, une par sommet autre que le départ. `arbre_largeur(G, 0)` renvoie `[0, 0, 0, 1, 2, 3, 4, -1, -1]`.



L'arbre du parcours en largeur depuis 0 : par exemple `pred[5]` vaut 3.

La liste `pred` contient, pour chaque sommet atteint, un chemin depuis le départ : il suffit de remonter de prédécesseur en prédécesseur.

4 Ce qu'on fait avec un parcours

Chaque problème de cette partie est le parcours précédent, avec une information en plus.

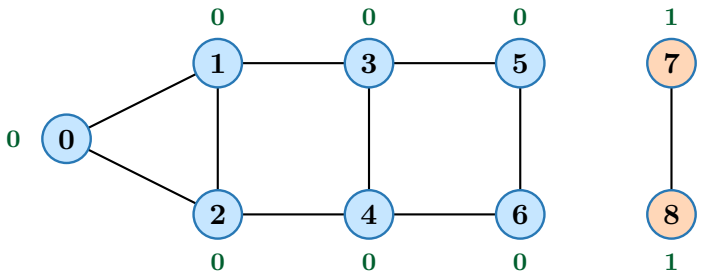
4.1 Connexité et composantes connexes

Un parcours depuis `s` marque exactement la composante connexe de `s`. Pour les trouver toutes, on relance un parcours depuis chaque sommet qui n'a pas encore été marqué, en donnant un nouveau numéro à chaque fois.

```python
def marquer_aux(G, s, numero, c):
    numero[s] = c
    for t in G[s]:
        if numero[t] == -1:
            marquer_aux(G, t, numero, c)

def composantes(G):
    """Renvoie (nb, numero) : nombre de composantes, numero[s] = celle de s."""
    numero = [-1 for s in range(len(G))]
    nb = 0
    for s in range(len(G)):
        if numero[s] == -1:
            marquer_aux(G, s, numero, nb)
            nb = nb + 1
    return nb, numero

def est_connexe(G):
    nb, numero = composantes(G)
    return nb == 1
```



`composantes(G)` renvoie `(2, [0, 0, 0, 0, 0, 0, 0, 1, 1])`.

L'information ajoutée est le numéro, qui sert aussi de marquage. La complexité est $O(n+m)$ **au total**, et non par parcours : chaque sommet n'est marqué qu'une fois, tous parcours confondus. Une fois `numero` calculé, on sait en $O(1)$ si deux sommets sont reliés par un chemin : `numero[s] == numero[t]`.

Sur une grille. C'est souvent sous cette forme que la question se présente (régions d'une image, domaines magnétiques) : on compte les groupes de cases égales à 1 qui se touchent. Le graphe n'est pas construit, on utilise `voisins_grille`, et une pile explicite : sur une grande grille, les appels récursifs imbriqués seraient trop nombreux pour Python.

1	1			2
	1		2	2
3		4	4	

Quatre domaines.

Comme pour `composantes`, une fonction auxiliaire marque un domaine entier et la fonction principale compte les départs.

```python
def marquer_domaine(grille, vus, i_depart, j_depart):
    """Marque toutes les cases du domaine de la case (i_depart, j_depart)."""
    pile = [(i_depart, j_depart)]
    while len(pile) > 0:
        (i, j) = pile.pop()
        if not vus[i][j]:
            vus[i][j] = True
            voisins = voisins_grille(len(grille), len(grille[0]), i, j)
            for (i_vois, j_vois) in voisins:
                if grille[i_vois][j_vois] == 1 and not vus[i_vois][j_vois]:
                    pile.append((i_vois, j_vois))

def nb_domaines(grille):
    """Nombre de groupes de cases égales à 1 qui se touchent par un côté."""
    nb_lignes = len(grille)
    nb_colonnes = len(grille[0])
    vus = [[False for j in range(nb_colonnes)] for i in range(nb_lignes)]
    nb = 0
    for i in range(nb_lignes):
        for j in range(nb_colonnes):
            if grille[i][j] == 1 and not vus[i][j]:
                marquer_domaine(grille, vus, i, j)
                nb = nb + 1
    return nb
```

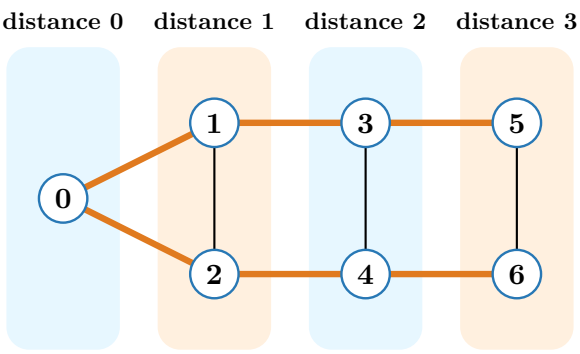
Si la grille a L lignes et C colonnes, le graphe a LC sommets et moins de $2LC$ arêtes : la complexité est $O(LC)$.

4.2 Distances et plus courts chemins en nombre d'arêtes

Le parcours en largeur visite les sommets **par distance croissante** au départ : d'abord ceux à distance 1, puis ceux à distance 2… L'information ajoutée est la distance, calculée au moment où le sommet entre dans la file.

```python
def distances(G, depart):
    """dist[t] : nombre minimal d'arêtes de depart à t (-1 si inaccessible)."""
    dist = [-1 for s in range(len(G))]
    pred = [-1 for s in range(len(G))]
    dist[depart] = 0
    pred[depart] = depart
    file = deque()
    file.append(depart)
    while len(file) > 0:
        s = file.popleft()
        for t in G[s]:
            if dist[t] == -1:
                dist[t] = dist[s] + 1
                pred[t] = s
                file.append(t)
    return dist, pred
```

Sur le graphe du cours, `distances(G, 0)` renvoie les distances `[0, 1, 1, 2, 2, 3, 3, -1, -1]`.



Le parcours en largeur depuis 0 avance niveau par niveau. Une arête relie deux sommets du même niveau ou de deux niveaux voisins.

Pourquoi c'est correct. Notons $\delta(t)$ le nombre minimal d'arêtes d'un chemin du départ à t.

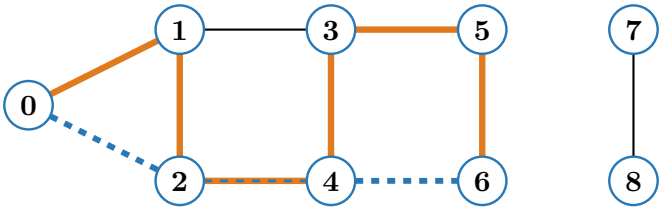
- La valeur `dist[t]` est la longueur d'un vrai chemin (celui que donnent les prédécesseurs), donc `dist[t]` $\geq \delta(t)$.
- Les sommets sortent de la file par valeurs de `dist` croissantes : à tout instant la file contient des sommets de valeur k, suivis de sommets de valeur $k+1$, puisqu'on sort un sommet de valeur k et qu'on ajoute à la fin des sommets de valeur $k+1$.
- Par récurrence sur k, tout sommet t tel que $\delta(t) = k$ reçoit `dist[t]` $\leq k$. En effet, t a un voisin u tel que $\delta(u) = k-1$, donc `dist[u]` $= k-1$ par hypothèse de récurrence. Quand u sort de la file, ou bien t est découvert à ce moment-là et reçoit la valeur k, ou bien il l'a déjà été, par un sommet sorti avant u, donc de valeur au plus $k-1$, et il a reçu une valeur au plus k.

Pour obtenir un plus court chemin, et pas seulement sa longueur, on remonte les prédécesseurs depuis l'arrivée, puis on retourne la liste :

```
def chemin(pred, depart, arrivee):
    """Chemin de depart à arrivee, lu dans pred (arrivee est accessible)."""
    sommets = [arrivee]
    s = arrivee
    while s != depart:
        s = pred[s]
        sommets.append(s)
    return sommets[::-1]
```

Avec `dist, pred = distances(G, 0)`, l'appel `chemin(pred, 0, 6)` renvoie `[0, 2, 4, 6]`. Si l'arrivée n'est pas accessible (`pred[arrivee]` vaut `-1`), la boucle ne termine pas : il faut le tester avant. Cette fonction `chemin` resservira telle quelle avec l'algorithme de Dijkstra.

Attention. Le parcours en profondeur trouve **un** chemin, pas un plus court : sur le graphe du cours il atteint 6 par $0, 1, 2, 4, 3, 5, 6$. Seul le parcours en largeur donne les distances.



De 0 à 6 : en trait plein le chemin du parcours en profondeur, en tirets celui du parcours en largeur. L'arête entre 2 et 4 est commune aux deux.

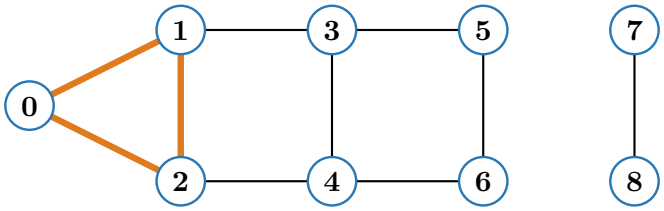
4.3 Détecter un cycle dans un graphe non orienté

Pendant un parcours en profondeur, tomber sur un voisin **déjà vu** signifie qu'on a deux chemins vers lui, donc un cycle. Il y a une exception : le sommet d'où l'on vient est toujours un voisin déjà vu, sans qu'il y ait de cycle. L'information ajoutée est donc le **parent** du sommet courant.

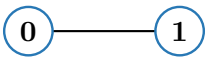
```
def cycle_aux(G, s, parent, vus):
    vus[s] = True
    for t in G[s]:
        if not vus[t]:
            if cycle_aux(G, t, s, vus):
                return True
        elif t != parent:
            return True
    return False

def a_un_cycle(G):
    """Le graphe non orienté G contient-il un cycle ?"""
    vus = [False for s in range(len(G))]
    for s in range(len(G)):
        if not vus[s]:
            if cycle_aux(G, s, -1, vus):
                return True
    return False
```

On relance le parcours depuis chaque sommet non vu, car le cycle peut se trouver dans n'importe quelle composante. Le graphe du cours a un cycle $(0, 1, 2)$: le parcours va de 0 à 1, de 1 à 2, et trouve en 2 le voisin 0, déjà vu, qui n'est pas son parent.



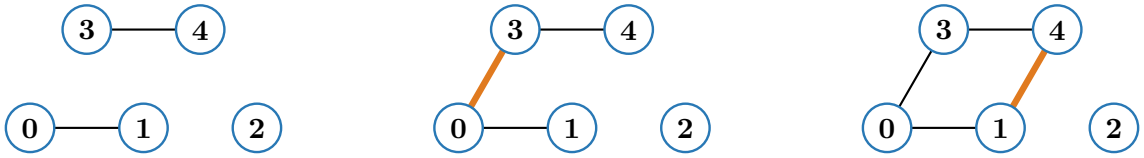
Un graphe réduit à une arête, `[[1], [0]]`, n'a pas de cycle, et c'est le test `t != parent` qui évite de se tromper sur ce cas : depuis 1, le voisin 0 est déjà vu, mais c'est le parent.



On peut aussi se ramener au décompte des composantes connexes, grâce au résultat suivant (issu de l'étude des arbres).

Remarque. Un graphe non orienté à n sommets, m arêtes et c composantes connexes vérifie $m \geq n - c$, avec égalité si et seulement s'il est sans cycle.

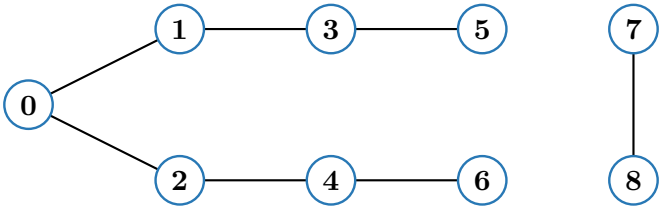
En effet, ajoutons les arêtes une à une en partant du graphe sans arête, qui a n composantes. Une arête qui relie deux composantes différentes en fait disparaître une et ne crée pas de cycle, une arête dont les deux extrémités sont déjà dans la même composante ferme un cycle et laisse le nombre de composantes inchangé. Il y a donc exactement $n - c$ arêtes du premier type, et le graphe est sans cycle si et seulement s'il n'y en a aucune du second.



$n = 5$, $m = 2$, $c = 3$.

$c = 2$, pas de cycle.

$c = 2$, un cycle.



Un graphe sans cycle : $n = 9$, $c = 2$ et $m = 7 = n - c$. Le graphe du cours a $m = 10 > 7$ arêtes : il a un cycle.

```python
def nb_aretes(G):
    """Nombre d'arêtes d'un graphe non orienté donné par listes d'adjacence."""
    total = 0
    for s in range(len(G)):
        total = total + len(G[s])
    return total // 2

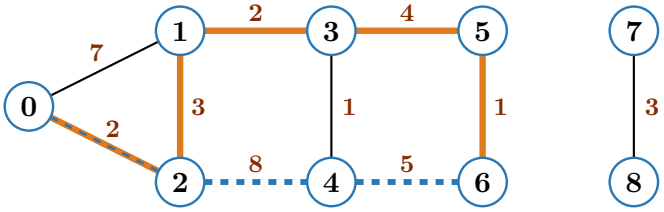
def a_un_cycle_comptage(G):
    nb, numero = composantes(G)
    return nb_aretes(G) > len(G) - nb
```

5 Plus courts chemins pondérés : l'algorithme de Dijkstra

5.1 Le problème

On se donne un graphe pondéré, orienté ou non, dont **tous les poids sont positifs ou nuls**, et un sommet de départ. On veut, pour chaque sommet t, la **distance** $\delta(t)$: le plus petit poids d'un chemin du départ à t (et $+\infty$ s'il n'y en a pas).

Le parcours en largeur ne convient plus : il minimise le nombre d'arêtes, pas le poids. Sur le graphe du cours, il va de 0 à 6 par $0, 2, 4, 6$, de poids $2 + 8 + 5 = 15$, alors que le chemin $0, 2, 1, 3, 5, 6$, plus long de deux arêtes, ne pèse que $2 + 3 + 2 + 4 + 1 = 12$.



De 0 à 6 : en tirets le chemin le plus court en nombre d'arêtes (poids 15), en trait plein le plus léger (poids 12). L'arête entre 0 et 2 est commune aux deux.

Une première idée. Si les poids sont des entiers strictement positifs, on peut couper chaque arête de poids p en p arêtes de poids 1, en ajoutant des sommets intermédiaires, puis lancer un parcours en largeur.



C'est correct, mais beaucoup trop coûteux quand les poids sont grands, et impossible s'ils ne sont pas entiers. On peut néanmoins garder cette intuition : le parcours en largeur est une **vague** qui part du départ et atteint les sommets par distance croissante. L'algorithme de Dijkstra calcule l'ordre dans lequel la vague atteint les **vrais** sommets, sans simuler les sommets intermédiaires.

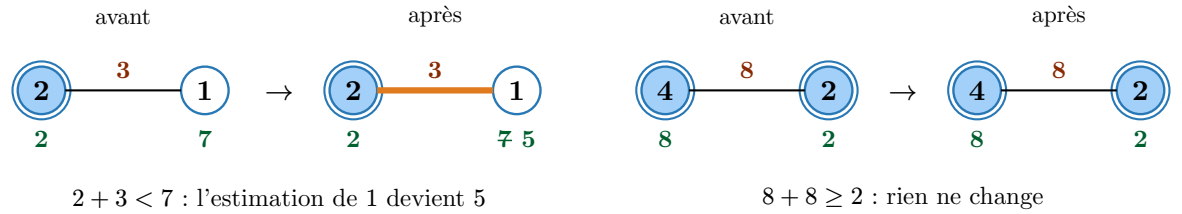
5.2 L'idée

On répartit les sommets en deux catégories.

- Les sommets **fixés** : ceux que la vague a déjà atteints. On connaît leur distance définitive.
- Les autres. Pour chacun, on tient à jour une **estimation** `dist[t]` : le poids du meilleur chemin connu jusqu'ici, c'est-à-dire du meilleur chemin qui ne passe que par des sommets fixés avant d'arriver à t. C'est un majorant de la vraie distance, qui vaut $+\infty$ tant qu'on ne connaît aucun chemin.

Au début, rien n'est fixé et la seule estimation finie est celle du départ, qui vaut 0. Puis on répète deux gestes.

1. **Choisir.** Parmi les sommets non fixés, on prend celui dont l'estimation est la plus petite, notons-le s, et on le fixe : on affirme que son estimation est sa vraie distance.
2. **Relâcher.** Pour chaque arête de s vers un voisin t, de poids p, on dispose d'un nouveau chemin vers t : aller en s, puis prendre l'arête. S'il est meilleur que ce qu'on connaissait, c'est-à-dire si `dist[s] + poids < dist[t]`, on remplace `dist[t]` par `dist[s] + poids`.



Deux relâchements de l'exécution qui suit. Sous chaque sommet, son estimation.

On s'arrête quand tous les sommets accessibles sont fixés.

C'est un algorithme glouton : le choix du sommet le plus proche n'est jamais remis en cause. Toute la question est de comprendre pourquoi ce choix est sûr.

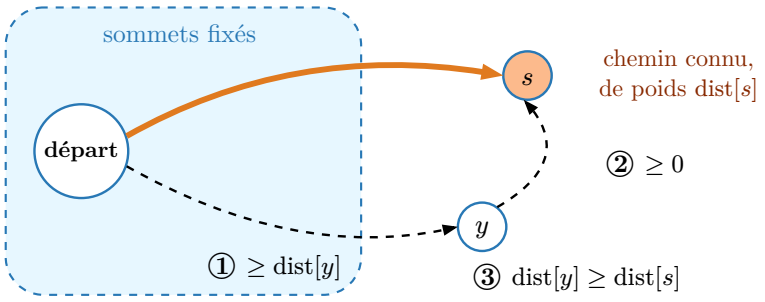
5.3 Pourquoi le choix est sûr

Précisons d'abord ce que contiennent les estimations.

Invariant. Avant chaque choix :
1. pour tout sommet fixé u, `dist[u]` est la distance $\delta(u)$,
2. pour tout sommet non fixé t, `dist[t]` est le plus petit poids d'un chemin du départ à t dont tous les sommets, sauf t, sont fixés (et $+\infty$ s'il n'y en a pas).

C'est vrai au début : rien n'est fixé, et le seul chemin dont tous les sommets sauf le dernier sont fixés est le chemin réduit au départ, de poids 0.

Le choix. Soit s le sommet non fixé d'estimation minimale. Son estimation est le poids d'un vrai chemin, donc $\delta(s) \leq \text{dist}[s]$. Il faut montrer qu'aucun chemin ne fait mieux.



poids du chemin en tirets $= ①+② \geq \text{dist}[y] \geq \text{dist}[s]$

Prenons un chemin **quelconque** du départ à s, et soit y son premier sommet non fixé (il y en a un, puisque s ne l'est pas, ce peut être s, ou le départ lui-même au tout premier choix).

- ①Le début du chemin, du départ à y, ne passe que par des sommets fixés avant d'arriver en y. Son poids est donc au moins `dist[y]`, d'après le point 2 de l'invariant.

- ②La suite du chemin, de y à s, a un poids **positif ou nul**. C'est ici, et seulement ici, que sert l'hypothèse sur les poids.
- ③Enfin `dist[y]` $\geq$ `dist[s]`, puisque s a été choisi d'estimation minimale parmi les sommets non fixés.

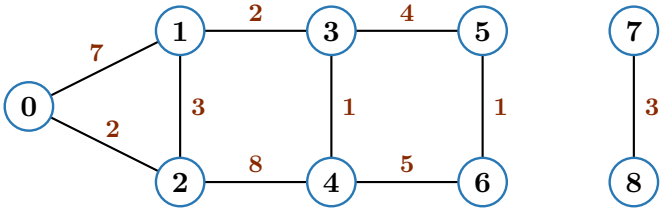
Le chemin pèse donc au moins `dist[s]`. Comme il était quelconque, $\delta(s) = \text{dist}[s]$: on peut fixer s, et le point 1 de l'invariant reste vrai.

Le relâchement. Il rétablit le point 2. Une fois s fixé, les nouveaux chemins « à travers la zone fixée » vers un sommet non fixé t sont ceux qui passent par s. Si un tel chemin se termine par une arête de s vers t, c'est celui que le relâchement examine. Sinon, il arrive en t par une arête issue d'un sommet u fixé plus tôt : il pèse au moins $\delta(u)$ plus le poids de cette arête, valeur déjà prise en compte quand on a relâché les arêtes de u.

> Les sommets sont fixés par distance croissante : c'est bien l'ordre d'arrivée de la vague. En effet, une estimation modifiée par le relâchement vaut `dist[s]` $+p \geq$ `dist[s]`, donc le sommet choisi ensuite est au moins aussi loin que s.

5.4 Exécution à la main

Appliquons l'algorithme au graphe ci-dessous, depuis le sommet 0.

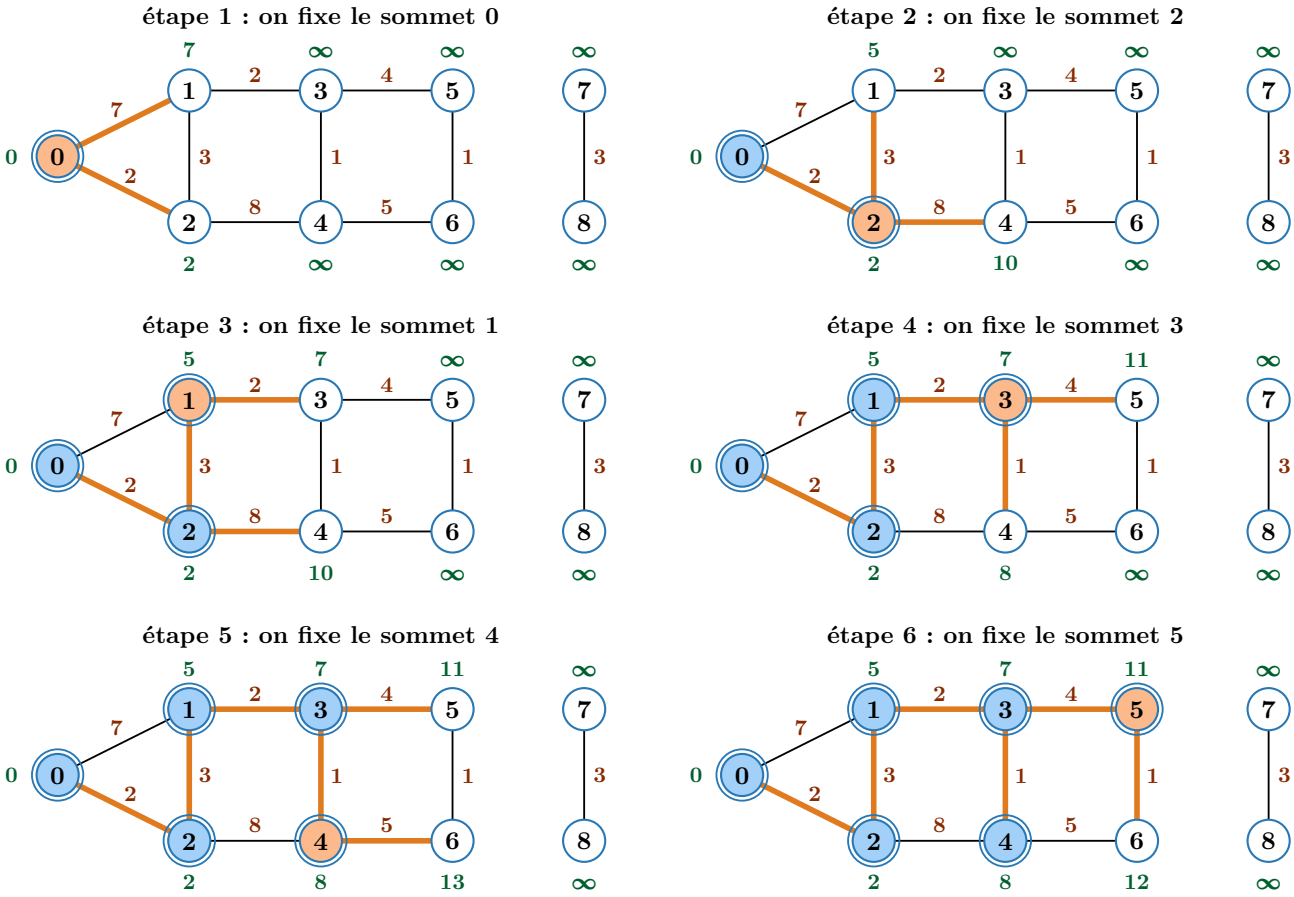


Chaque ligne du tableau donne les estimations après avoir fixé un sommet et relâché ses arêtes, une valeur encadrée est définitive, une valeur en gras vient d'être améliorée.

Sommet fixé	0	1	2	3	4	5	6	7	8
(début)	0	∞	∞	∞	∞	∞	∞	∞	∞
0	$\boxed{0}$	**7**	**2**	∞	∞	∞	∞	∞	∞
2	$\boxed{0}$	**5**	$\boxed{2}$	∞	**10**	∞	∞	∞	∞
1	$\boxed{0}$	$\boxed{5}$	$\boxed{2}$	**7**	10	∞	∞	∞	∞
3	$\boxed{0}$	$\boxed{5}$	$\boxed{2}$	$\boxed{7}$	**8**	**11**	∞	∞	∞
4	$\boxed{0}$	$\boxed{5}$	$\boxed{2}$	$\boxed{7}$	$\boxed{8}$	11	**13**	∞	∞
5	$\boxed{0}$	$\boxed{5}$	$\boxed{2}$	$\boxed{7}$	$\boxed{8}$	$\boxed{11}$	**12**	∞	∞
6	$\boxed{0}$	$\boxed{5}$	$\boxed{2}$	$\boxed{7}$	$\boxed{8}$	$\boxed{11}$	$\boxed{12}$	∞	∞

Trois moments à regarder de près.

- Après avoir fixé 0, l'estimation de 1 vaut 7 (l'arête directe). En fixant 2, on découvre le chemin $0, 2, 1$ de poids $2 + 3 = 5$: l'estimation est améliorée. Une estimation n'est définitive que lorsque le sommet est fixé.
- L'estimation de 4 passe de 10 à 8, celle de 6 de 13 à 12 : même chose.
- À la fin, les sommets 7 et 8 ont toujours une estimation infinie : ils ne sont pas accessibles depuis 0, et l'algorithme s'arrête.



Les six premières étapes (la septième fixe le sommet 6 sans rien modifier). Fond coloré et double contour : sommets fixés, le dernier fixé en orange. À côté de chaque sommet, son estimation, sur les arêtes, les poids, en gras, l'arête par laquelle passe le meilleur chemin connu.

5.5 Le code

On range les estimations dans une liste `dist` et l'on note dans une liste de booléens `fixe` les sommets fixés. Le choix du sommet le plus proche se fait par une simple recherche de minimum.

```
def plus_proche(dist, fixe):
    """Sommet non fixé d'estimation minimale (-1 s'il n'y a que des infinis)."""
    s = -1
    mini = INFINI
    for t in range(len(dist)):
        if not fixe[t] and dist[t] < mini:
            s = t
            mini = dist[t]
    return s
```

```python
def dijkstra(G, depart):
    """dist[t] : distance de depart à t (INFINI si t est inaccessible)."""
    n = len(G)
    dist = [INFINI for s in range(n)]
    fixe = [False for s in range(n)]
    dist[depart] = 0
    s = plus_proche(dist, fixe)
    while s != -1:
        fixe[s] = True                          # choisir : s est définitif
        for (t, poids) in G[s]:                 # relâcher les arêtes issues de s
            if dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
        s = plus_proche(dist, fixe)
    return dist
```

`dijkstra(G_POIDS, 0)` renvoie `[0, 5, 2, 7, 8, 11, 12, inf, inf]`, la dernière ligne du tableau.

Quelques remarques sur ce code.

- Au premier passage, `plus_proche` renvoie le départ, seul sommet d'estimation finie.
- `plus_proche` renvoie `-1` quand il ne reste que des sommets à l'infini : ils sont inaccessibles, la boucle s'arrête. Comme `mini` vaut `INFINI` au début et que le test est strict, un sommet à l'infini n'est jamais choisi.
- Le relâchement ne teste pas si `t` est déjà fixé : c'est inutile, car un sommet fixé avant `s` vérifie `dist[t] <= dist[s]`, donc `dist[s] + poids < dist[t]` est faux dès que le poids est positif ou nul.
- C'est le schéma général de parcours : le sac est l'ensemble des sommets non fixés d'estimation finie, et l'on en sort le plus proche du départ.

Terminaison. Chaque tour de boucle fixe un sommet qui ne l'était pas : il y a au plus n tours.

Complexité. Il y a au plus n tours, et chacun appelle `plus_proche`, en $O(n)$: cela fait $O(n^2)$. Les relâchements coûtent $O(d(s))$ pour le sommet `s`, soit $O(m)$ en tout. Comme $m \leq n^2$, la complexité est

$$O(n^2 + m) = O(n^2).$$

> La recherche du minimum est ce que le programme appelle une **file de priorité naïve**. Avec une vraie file de priorité, qu'un sujet fournirait avec sa documentation et dont les opérations coûtent $O(\ln(n))$, la complexité devient $O((n+m)\ln(n))$: c'est mieux quand le graphe a peu d'arêtes, et le logarithme ne doit pas être oublié.

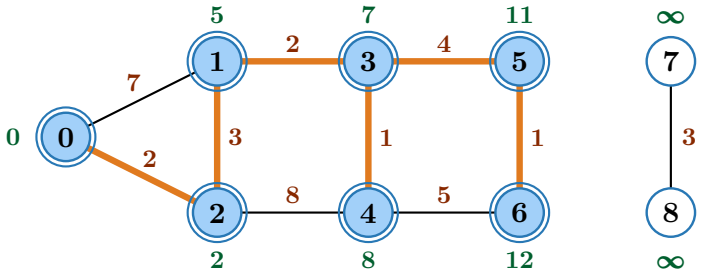
5.6 Variantes

Ce sont de petites modifications du même code, qu'il faut savoir faire à la demande.

Retrouver les chemins. Comme pour le parcours en largeur, on retient le prédécesseur : chaque fois qu'une estimation est améliorée, le meilleur chemin connu vers `t` arrive par `s`.

```python
def dijkstra_pred(G, depart):
    """Renvoie (dist, pred) : pred[t] précède t sur un plus court chemin."""
    n = len(G)
    dist = [INFINI for s in range(n)]
    pred = [-1 for s in range(n)]
    fixe = [False for s in range(n)]
    dist[depart] = 0
    pred[depart] = depart
    s = plus_proche(dist, fixe)
    while s != -1:
        fixe[s] = True
        for (t, poids) in G[s]:
            if dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
                pred[t] = s
        s = plus_proche(dist, fixe)
    return dist, pred
```

Sur le graphe du cours, `pred` vaut `[0, 2, 0, 1, 3, 3, 5, -1, -1]`, et la fonction `chemin` de la partie précédente donne `chemin(pred, 0, 6) = [0, 2, 1, 3, 5, 6]`. Les arêtes `(pred[t], t)` forment encore un arbre, l'arbre des plus courts chemins.



L'arbre des plus courts chemins depuis 0, et les distances.

S'arrêter à la cible. Si un seul sommet d'arrivée nous intéresse, on peut s'arrêter dès qu'il est choisi, puisque son estimation est alors définitive.

```python
def dijkstra_cible(G, depart, arrivee):
    """Poids minimal d'un chemin de depart à arrivee."""
    n = len(G)
    dist = [INFINI for s in range(n)]
    fixe = [False for s in range(n)]
    dist[depart] = 0
    s = plus_proche(dist, fixe)
    while s != -1 and s != arrivee:
        fixe[s] = True
        for (t, poids) in G[s]:
            if dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
        s = plus_proche(dist, fixe)
    return dist[arrivee]
```

La complexité dans le pire cas ne change pas, mais on ne fixe que des sommets au plus aussi loin du départ que l'arrivée.

Avec une matrice. Si `M[s][t]` vaut `INFINI` en l'absence d'arête, on peut relâcher tous les sommets sans distinction, puisque `dist[s] + INFINI` n'améliore jamais rien :

```python
def dijkstra_matrice(M, depart):
    """M[s][t] : poids de l'arête de s à t, INFINI s'il n'y en a pas."""
    n = len(M)
    dist = [INFINI for s in range(n)]
    fixe = [False for s in range(n)]
    dist[depart] = 0
    s = plus_proche(dist, fixe)
    while s != -1:
        fixe[s] = True
        for t in range(n):
            if dist[s] + M[s][t] < dist[t]:
                dist[t] = dist[s] + M[s][t]
        s = plus_proche(dist, fixe)
    return dist
```

La complexité est toujours $O(n^2)$: avec une matrice, la version naïve est la bonne.

Sur un graphe qu'on ne construit pas. Les sommets sont les cases d'une grille, et entrer dans une case coûte la valeur qui y est inscrite. Le graphe est donné par une fonction `successeurs`, et les sommets sont des couples : les listes `dist` et `fixe` deviennent des dictionnaires, qui ne contiennent que les sommets déjà découverts.

```python
AUCUNE = (-1, -1)

def successeurs(grille, case):
    """Liste des couples (case voisine, coût pour y entrer)."""
    (i, j) = case
    resultat = []
    for (i_vois, j_vois) in voisins_grille(len(grille), len(grille[0]), i, j):
        resultat.append(((i_vois, j_vois), grille[i_vois][j_vois]))
    return resultat

def plus_proche_dico(dist, fixe):
    """Case non fixée d'estimation minimale (AUCUNE s'il n'y en a plus)."""
    s = AUCUNE
    mini = INFINI
    for t in dist:
        if not (t in fixe) and dist[t] < mini:
            s = t
            mini = dist[t]
    return s
```

```python
def dijkstra_grille(grille, depart, arrivee):
    """Coût minimal d'un chemin de la case depart à la case arrivee."""
    dist = {depart: 0}      # estimations des cases déjà découvertes
    fixe = {}               # fixe[s] = True pour les cases fixées
    s = plus_proche_dico(dist, fixe)
    while s != AUCUNE and s != arrivee:
        fixe[s] = True
        for (t, poids) in successeurs(grille, s):
            if not (t in dist) or dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
        s = plus_proche_dico(dist, fixe)
    if arrivee in dist:
        return dist[arrivee]
    return INFINI
```

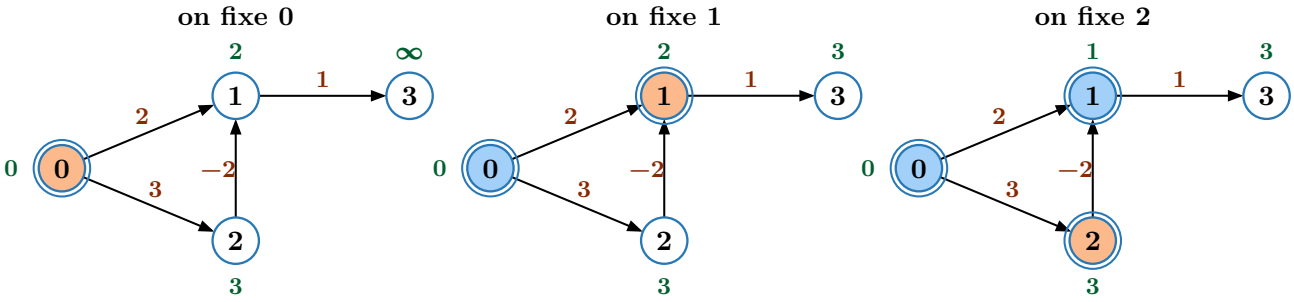
Un sommet absent de `dist` a une estimation infinie : c'est le sens du test `not (t in dist)`. La constante `AUCUNE`, qui n'est pas une case de la grille, joue le rôle du `-1` des versions précédentes.

1	9	**1**	**1**
1	9	**1**	9
1	**1**	**1**	1

De la case $(0,0)$ à la case $(0,3)$: le meilleur chemin contourne les cases de coût 9 et coûte 7.

5.7 Les limites de l'algorithme

Attention. Avec un poids négatif, l'algorithme de Dijkstra peut renvoyer un résultat faux, sans aucun message d'erreur.

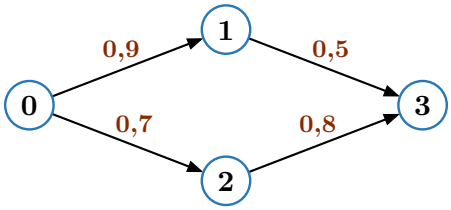


Dans ce graphe orienté, l'algorithme fixe 0, puis 1 avec la distance 2, et annonce donc 3 pour le sommet 3. Il fixe ensuite 2 et découvre le chemin $0, 2, 1$ de poids $3 - 2 = 1$: trop tard, le sommet 1 avait été fixé à tort, il ne sera plus choisi, et l'estimation du sommet 3 reste 3 alors que sa vraie distance est 2. L'argument de sûreté ne tient plus : la fin d'un chemin peut maintenant **faire baisser** son poids.

Se ramener à des poids positifs. Beaucoup de problèmes ne sont pas donnés comme une somme à minimiser. Supposons que chaque arête porte une probabilité $q \in]0,1]$ et que l'on cherche le chemin dont le **produit** des probabilités est **maximal**. On se ramène à Dijkstra en donnant à chaque arête le poids $-\ln q$, qui est positif ou nul : comme

$$-\ln(q_1 q_2 \cdots q_k) = (-\ln q_1) + (-\ln q_2) + \cdots + (-\ln q_k),$$

maximiser le produit revient à minimiser la somme des poids.

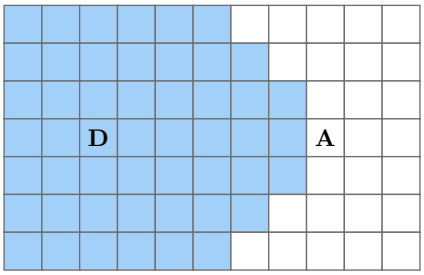


De 0 à 3 : par 1, la probabilité est $0,9 \times 0,5 = 0,45$, par 2, elle est $0,7 \times 0,8 = 0,56$.

5.8 L'algorithme A* (HP)

Quand on cherche un chemin vers **une** cible, l'algorithme de Dijkstra ne sait pas où elle se trouve : il fixe les sommets par distance croissante au départ, donc tous ceux qui sont plus proches du départ que la cible, dans toutes les directions. Sur une grille 7×11 dont toutes les cases coûtent 1, pour aller de la case D à la case A, il fixe 50 cases, y compris toutes celles qui sont à gauche de D.

Dijkstra : 50 cases fixées



En bleu, les cases fixées par l'algorithme de Dijkstra avant d'atteindre A.

L'idée. On se donne une **heuristique** : une fonction h qui estime, pour chaque sommet t, ce qu'il reste à parcourir de t jusqu'à l'arrivée. Au lieu de choisir le sommet non fixé qui minimise `dist[t]`, on choisit celui qui minimise

$$\text{dist}[t] + h(t),$$

c'est-à-dire le poids estimé d'un chemin complet du départ à l'arrivée **passant par** t. Rien d'autre ne change : ni les estimations `dist`, ni le relâchement, ni l'arrêt quand l'arrivée est choisie.

Un exemple d'heuristique. Sur une grille où l'on se déplace d'une case à une case voisine, il faut au moins $|i_1 - i_2| + |j_1 - j_2|$ déplacements pour aller de la case (i_1, j_1) à la case (i_2, j_2) : c'est la **distance de Manhattan**. Si entrer dans une case coûte au moins 1, c'est une estimation, par défaut, de ce qu'il reste à payer. Sur une carte routière, on prendrait de même la distance à vol d'oiseau.

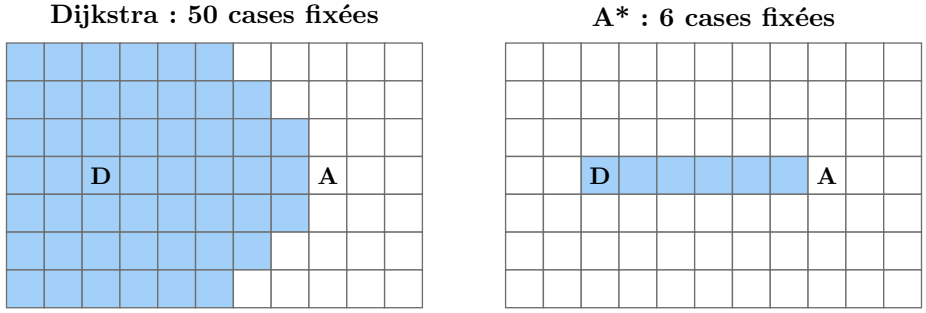
```python
def ecart(x, y):
    if x < y:
        return y - x
    return x - y

def manhattan(case1, case2):
    """Nombre minimal de déplacements pour aller de case1 à case2."""
    (i1, j1) = case1
    (i2, j2) = case2
    return ecart(i1, i2) + ecart(j1, j2)

def plus_prometteur(dist, fixe, arrivee):
    """Case non fixée qui minimise dist[t] + manhattan(t, arrivee)."""
    s = AUCUNE
    mini = INFINI
    for t in dist:
        if not (t in fixe) and dist[t] + manhattan(t, arrivee) < mini:
            s = t
            mini = dist[t] + manhattan(t, arrivee)
    return s
```

```python
def a_etoile_grille(grille, depart, arrivee):
    """Coût minimal de depart à arrivee (chaque case coûte au moins 1)."""
    dist = {depart: 0}
    fixe = {}
    s = plus_prometteur(dist, fixe, arrivee)
    while s != AUCUNE and s != arrivee:
        fixe[s] = True
        for (t, poids) in successeurs(grille, s):
            if not (t in dist) or dist[s] + poids < dist[t]:
                dist[t] = dist[s] + poids
        s = plus_prometteur(dist, fixe, arrivee)
    if arrivee in dist:
        return dist[arrivee]
    return INFINI
```

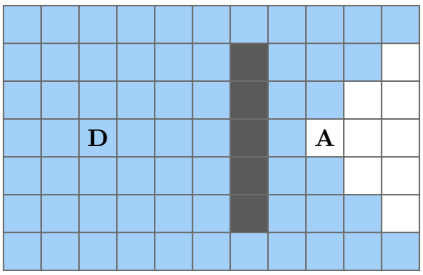
C'est `dijkstra_grille`, où `plus_proche_dico` est remplacée par `plus_prometteur`. Sur la grille précédente, A* ne fixe plus que les 6 cases qui mènent droit à A :



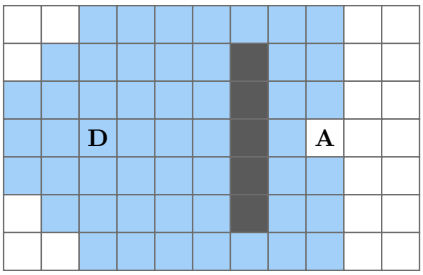
En bleu, les cases fixées avant d'atteindre A. Les deux algorithmes trouvent le même coût, 6.

Ce cas est idéal. Si un mur de cases très coûteuses (en gris foncé, coût 20) sépare D de A, l'heuristique se trompe sur la direction à prendre et le gain est bien moindre :

Dijkstra : 63 cases fixées



A* : 51 cases fixées



Avec un mur : même coût, 12, en contournant le mur.

Quelle heuristique choisir ? Notons reste(t) la distance d'un sommet t à l'arrivée : ce qu'il reste **réellement** à parcourir.

Définition. Une heuristique h est **admissible** si elle ne surestime jamais ce qu'il reste à parcourir : $h(t) \leq \text{reste}(t)$ pour tout sommet t.

Elle est **cohérente** si h est nulle à l'arrivée et si, pour toute arête de s vers t de poids p, on a $h(s) \leq p + h(t)$: en suivant une arête, l'estimation ne baisse pas plus que ce que l'on vient de payer.

Une heuristique cohérente est admissible : en ajoutant les inégalités le long d'un plus court chemin de t à l'arrivée, on obtient $h(t) \leq \text{reste}(t) + 0$.

Propriété. Si l'heuristique est cohérente, l'algorithme A* renvoie la distance du départ à l'arrivée.

Preuve. Remplaçons le poids p de chaque arête de s vers t par $\tilde{p} = p + h(t) - h(s)$, qui est positif ou nul puisque h est cohérente. Le long d'un chemin du départ à un sommet t, les termes en h se simplifient de proche en proche : son nouveau poids est l'ancien, augmenté de $h(t) - h(\text{départ})$. Pour un sommet t donné, tous les chemins sont donc décalés de la même quantité, et les plus courts chemins sont les mêmes. Or l'algorithme de Dijkstra sur les poids $\tilde{p}$ choisit le sommet qui minimise $\text{dist}[t] + h(t) - h(\text{départ})$, où $h(\text{départ})$ est une constante : c'est exactement le choix de A*. Les relâchements sont aussi les mêmes, car la comparaison de deux chemins vers un même sommet n'est pas modifiée par le décalage. Ainsi A* **est** l'algorithme de Dijkstra sur un graphe à poids positifs, et il est correct.

La distance de Manhattan est cohérente quand chaque case coûte au moins 1 : un déplacement la fait varier d'exactement 1, donc elle ne baisse jamais de plus que le coût payé. La distance à vol d'oiseau l'est aussi, par l'inégalité triangulaire.

Ce que l'on gagne dépend de la qualité de l'estimation.

- Avec $h = 0$, A* est l'algorithme de Dijkstra.
- Plus h est proche de la vraie distance restante, tout en restant en dessous, moins on fixe de sommets. Dans la figure, la distance de Manhattan **est** la vraie distance, et l'on ne fixe que les cases d'un plus court chemin.
- Si h surestime, le résultat peut être faux. Dans la grille ci-dessous, de la case en bas à gauche à la case en bas à droite, le meilleur chemin contourne la case de coût 9 et coûte 4, avec l'heuristique $5 \times$ Manhattan, l'algorithme fonce tout droit et renvoie 10.

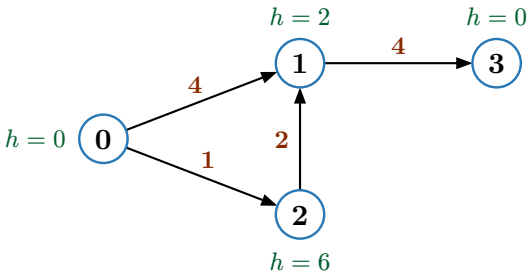
1	**1**	**1**
1	9	**1**

Le meilleur chemin : coût 4.

1	1	1
1	**9**	**1**

Le chemin trouvé avec $5 \times$ Manhattan : coût 10.

On demande souvent seulement que l'heuristique soit **admissible**. C'est la condition qui garantit le résultat lorsqu'on accepte de revenir sur un sommet déjà fixé si son estimation s'améliore, avec notre code, où un sommet fixé n'est jamais choisi une seconde fois, c'est la cohérence qui convient. Les heuristiques naturelles, comme les deux précédentes, vérifient les deux.



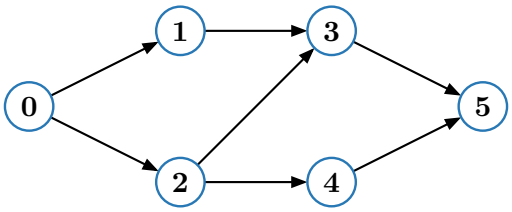
Une heuristique admissible mais pas cohérente ($h(2) = 6 > 2 + h(1)$) : on fixe 1 avant 2 et l'on obtient 8 pour aller de 0 à 3, au lieu de 7.

5.9 Quel algorithme pour quel problème ?

Algorithme	Hypothèses	Complexité	Ce qu'il calcule
Parcours en largeur	graphe non pondéré	$O(n+m)$	distances en nombre d'arêtes depuis un sommet
Dijkstra	poids positifs ou nuls	$O(n^2)$, ou $O((n+m)\ln(n))$ avec une file de priorité	distances depuis un sommet
A*	poids positifs ou nuls, heuristique cohérente	comme Dijkstra dans le pire cas	distance d'un sommet à une cible, en fixant moins de sommets
Programmation dynamique	graphe orienté sans cycle, poids quelconques	$O(n+m)$	distances depuis un sommet, ou vers un sommet

6 Graphes orientés sans cycle

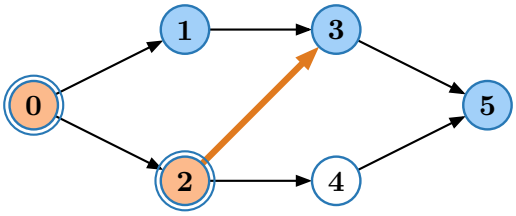
Les graphes orientés sans cycle sont partout : tâches dont certaines doivent être faites avant d'autres, dépendances entre des calculs, sous-problèmes d'une programmation dynamique. Les deux algorithmes de cette partie ne sont pas au programme, mais ce sont deux variantes immédiates du parcours en profondeur.



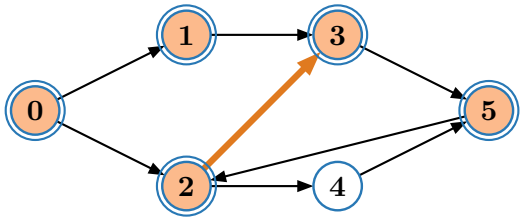
6.1 Détecter un cycle (HP)

Dans un graphe orienté, tomber sur un sommet déjà vu ne prouve rien : dans le graphe ci-dessus, le parcours depuis 0 atteint 3 par 1, puis le retrouve depuis 2, sans qu'il y ait de cycle. Il y a un cycle seulement si l'on retombe sur un sommet dont l'exploration est **en cours**, c'est-à-dire un sommet de la pile des appels. On distingue donc trois états.

- **non vu** : le parcours n'y est pas encore passé,
- **en cours** : son exploration a commencé et n'est pas terminée, il est dans la pile des appels,
- **fini** : son exploration est terminée.



Sans cycle. Quand 2 examine son voisin 3, celui-ci est **fini** : pas de cycle.



Avec une arête de 5 vers 2. Quand 2 examine 3, celui-ci est **en cours** : cycle $3, 5, 2$.

Dans les deux figures, les sommets en cours ont un double contour et un fond orange, les sommets finis un fond bleu, et le sommet 4 n'est pas encore vu. À gauche, le parcours depuis 0 a exploré $1, 3, 5$, est revenu en 0, et explore 2 : la pile des appels est $0, 2$. À droite, il est descendu $0, 1, 3, 5, 2$ sans jamais remonter : tous ces sommets sont en cours, et l'arête de 2 vers 3 referme un cycle.

```
NON_VU = 0
EN_COURS = 1
FINI = 2

def cycle_oriente_aux(G, s, etat):
    etat[s] = EN_COURS
    for t in G[s]:
        if etat[t] == EN_COURS:
            return True
        if etat[t] == NON_VU:
            if cycle_oriente_aux(G, t, etat):
                return True
    etat[s] = FINI
    return False

def a_un_cycle_oriente(G):
    """Le graphe orienté G contient-il un cycle ?"""
    etat = [NON_VU for s in range(len(G))]
    for s in range(len(G)):
        if etat[s] == NON_VU:
            if cycle_oriente_aux(G, s, etat):
                return True
    return False
```

6.2 Ordre topologique (HP)

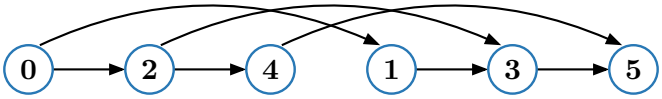
Définition. Un **ordre topologique** d'un graphe orienté est une façon de ranger tous ses sommets sur une ligne de sorte que toutes les arêtes aillent de la gauche vers la droite.

Il en existe un si et seulement si le graphe n'a pas de cycle (le long d'un cycle, on ne peut pas aller toujours vers la droite). Pour le construire, on note chaque sommet au moment où son exploration **se termine** : à cet instant, tout ce qui est accessible depuis lui est déjà noté. Il suffit de retourner la liste obtenue.

```
def sortie_aux(G, s, vus, sortis):
    vus[s] = True
    for t in G[s]:
        if not vus[t]:
            sortie_aux(G, t, vus, sortis)
    sortis.append(s)
```

```python
def ordre_topologique(G):
    """Un ordre topologique du graphe orienté sans cycle G."""
    vus = [False for s in range(len(G))]
    sortis = []
    for s in range(len(G)):
        if not vus[s]:
            sortie_aux(G, s, vus, sortis)
    return sortis[::-1]
```

Sur le graphe de la figure, `G_ORIENTE = [[1, 2], [3], [3, 4], [5], [5], []]`, on obtient `[0, 2, 4, 1, 3, 5]`.



Pourquoi c'est correct. Soit une arête de s vers t. Si t n'est pas encore vu quand on explore s, il est exploré pendant l'exploration de s et sort donc avant lui. S'il est déjà vu, il est déjà sorti, car sinon son exploration serait en cours et l'on aurait un cycle. Dans les deux cas t sort avant s, donc se retrouve à sa droite une fois la liste retournée.

6.3 Le lien avec la programmation dynamique

Dans une programmation dynamique, les états sont les sommets d'un graphe orienté sans cycle, avec une arête de chaque état vers ceux dont il dépend.

- **Remplir un tableau de bas en haut**, c'est traiter les états dans un ordre où chacun vient après ceux dont il dépend : l'ordre de remplissage est un ordre topologique (retourné) de ce graphe.
- **Mémoïser**, c'est faire un parcours en profondeur de ce graphe : le dictionnaire `memo` joue le rôle du marquage, et la valeur d'un état est calculée quand son exploration se termine.

Par exemple, compter les chemins entre deux sommets d'un graphe orienté sans cycle est un parcours en profondeur mémoïsé :

```python
def nb_chemins_aux(G, s, arrivee, memo):
    if s == arrivee:
        return 1
    if s in memo:
        return memo[s]
    total = 0
    for t in G[s]:
        total = total + nb_chemins_aux(G, t, arrivee, memo)
    memo[s] = total
    return total

def nb_chemins(G, depart, arrivee):
    """Nombre de chemins de depart à arrivee (G orienté sans cycle)."""
    return nb_chemins_aux(G, depart, arrivee, {})
```

Sans le dictionnaire, le nombre d'appels peut être exponentiel, avec lui, la complexité est $O(n+m)$. `nb_chemins(G_ORIENTE, 0, 5)` vaut 3 : à côté de chaque sommet, le nombre de chemins qui en partent vers 5, somme des nombres de ses voisins.

